```
///////////////////////////////////////////////////////////////////////////
// C++ source code of StockfishEngine.cpp Version 17.0 at greenchess.com //
///////////////////////////////////////////////////////////////////////////

// import needed system files //
#include "StockfishEngine.h"
#include <iostream>
#include <algorithm>
#include <cassert>
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <cstdio>
#include <iosfwd>
#include <optional>
#include <string>
#include <vector>
    #include <cassert>
    #include <cstdint>
#include <memory>
#include <new>
#include <type_traits>
#include <utility>
#include <atomic>
#include <cstdlib>
#include <limits>
#include <map>
#include <mutex>
#include <set>
#include <sstream>
#include <thread>
#include <cstring>
    #include <sched.h>
#include <variant>
#include <limits.h>
#include <type_traits>  // IWYU pragma: keep
#include <array>
#include <cmath>
#include <deque>
#include <functional>
#include <algorithm>  // IWYU pragma: keep
#include <string_view>
    #include <pthread.h>
    #include <functional>
#include <condition_variable>
#include <tuple>
#include <fstream>
#include <initializer_list>
#include <sys/stat.h>
    #include <fcntl.h>
    #include <sys/mman.h>
    #include <unistd.h>
#include <cctype>
#include <iomanip>
#include <bitset>
#include <iterator>
#include <list>
#include <ratio>
#include <unordered_map>
    #include <stdlib.h>
    #include <ios>       // std::hex, std::dec
    #include <iostream>  // std::cerr
    #include <ostream>   // std::endl
constexpr std::string_view PieceToChar(" PNBRQK  pnbrqk");
constexpr auto StartFEN = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
bool StockfishIsDone = false;
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef STOCKFISH_SIMD_H_INCLUDED
#define STOCKFISH_SIMD_H_INCLUDED

#if defined(USE_AVX2)

#elif defined(USE_SSE41)

#elif defined(USE_SSSE3)

#elif defined(USE_SSE2)

#elif defined(USE_NEON)
#endif

namespace Stockfish::Simd {

#if defined(USE_AVX512)

[[maybe_unused]] static int m512_hadd(__m512i sum, int bias) {
    return _mm512_reduce_add_epi32(sum) + bias;
}

[[maybe_unused]] static void m512_add_dpbusd_epi32(__m512i& acc, __m512i a, __m512i b) {

    #if defined(USE_VNNI)
    acc = _mm512_dpbusd_epi32(acc, a, b);
    #else
    __m512i product0 = _mm512_maddubs_epi16(a, b);
    product0         = _mm512_madd_epi16(product0, _mm512_set1_epi16(1));
    acc              = _mm512_add_epi32(acc, product0);
    #endif
}

#endif

#if defined(USE_AVX2)

[[maybe_unused]] static int m256_hadd(__m256i sum, int bias) {
    __m128i sum128 = _mm_add_epi32(_mm256_castsi256_si128(sum), _mm256_extracti128_si256(sum, 1));
    sum128         = _mm_add_epi32(sum128, _mm_shuffle_epi32(sum128, _MM_PERM_BADC));
    sum128         = _mm_add_epi32(sum128, _mm_shuffle_epi32(sum128, _MM_PERM_CDAB));
    return _mm_cvtsi128_si32(sum128) + bias;
}

[[maybe_unused]] static void m256_add_dpbusd_epi32(__m256i& acc, __m256i a, __m256i b) {

    #if defined(USE_VNNI)
    acc = _mm256_dpbusd_epi32(acc, a, b);
    #else
    __m256i product0 = _mm256_maddubs_epi16(a, b);
    product0         = _mm256_madd_epi16(product0, _mm256_set1_epi16(1));
    acc              = _mm256_add_epi32(acc, product0);
    #endif
}

#endif

#if defined(USE_SSSE3)

[[maybe_unused]] static int m128_hadd(__m128i sum, int bias) {
    sum = _mm_add_epi32(sum, _mm_shuffle_epi32(sum, 0x4E));  //_MM_PERM_BADC
    sum = _mm_add_epi32(sum, _mm_shuffle_epi32(sum, 0xB1));  //_MM_PERM_CDAB
    return _mm_cvtsi128_si32(sum) + bias;
}

[[maybe_unused]] static void m128_add_dpbusd_epi32(__m128i& acc, __m128i a, __m128i b) {

    __m128i product0 = _mm_maddubs_epi16(a, b);
    product0         = _mm_madd_epi16(product0, _mm_set1_epi16(1));
    acc              = _mm_add_epi32(acc, product0);
}

#endif

#if defined(USE_NEON_DOTPROD)

[[maybe_unused]] static void
dotprod_m128_add_dpbusd_epi32(int32x4_t& acc, int8x16_t a, int8x16_t b) {

    acc = vdotq_s32(acc, a, b);
}
#endif

#if defined(USE_NEON)

[[maybe_unused]] static int neon_m128_reduce_add_epi32(int32x4_t s) {
    #if USE_NEON >= 8
    return vaddvq_s32(s);
    #else
    return s[0] + s[1] + s[2] + s[3];
    #endif
}

[[maybe_unused]] static int neon_m128_hadd(int32x4_t sum, int bias) {
    return neon_m128_reduce_add_epi32(sum) + bias;
}

#endif

#if USE_NEON >= 8
[[maybe_unused]] static void neon_m128_add_dpbusd_epi32(int32x4_t& acc, int8x16_t a, int8x16_t b) {

    int16x8_t product0 = vmull_s8(vget_low_s8(a), vget_low_s8(b));
    int16x8_t product1 = vmull_high_s8(a, b);
    int16x8_t sum      = vpaddq_s16(product0, product1);
    acc                = vpadalq_s16(acc, sum);
}
#endif
}

#endif  // STOCKFISH_SIMD_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef MISC_H_INCLUDED
#define MISC_H_INCLUDED

#define stringify2(x) #x
#define stringify(x) stringify2(x)

namespace Stockfish {

std::string engine_info(bool to_uci = false);
std::string compiler_info();

// Preloads the given address in L1/L2 cache. This is a non-blocking
// function that doesn't stall the CPU waiting for data to be loaded from memory,
// which can be quite slow.
void prefetch(const void* addr);

void start_logger(const std::string& fname);

size_t str_to_size_t(const std::string& s);

#if defined(__linux__)

struct PipeDeleter {
    void operator()(FILE* file) const {
        if (file != nullptr)
        {
            pclose(file);
        }
    }
};

#endif

// Reads the file as bytes.
// Returns std::nullopt if the file does not exist.
std::optional<std::string> read_file_to_string(const std::string& path);

void dbg_hit_on(bool cond, int slot = 0);
void dbg_mean_of(int64_t value, int slot = 0);
void dbg_stdev_of(int64_t value, int slot = 0);
void dbg_extremes_of(int64_t value, int slot = 0);
void dbg_correl_of(int64_t value1, int64_t value2, int slot = 0);
void dbg_print();

using TimePoint = std::chrono::milliseconds::rep;  // A value in milliseconds
static_assert(sizeof(TimePoint) == sizeof(int64_t), "TimePoint should be 64 bits");
inline TimePoint now() {
    return std::chrono::duration_cast<std::chrono::milliseconds>(
             std::chrono::steady_clock::now().time_since_epoch())
      .count();
}

inline std::vector<std::string> split(const std::string& s, const std::string& delimiter) {
    std::vector<std::string> res;

    if (s.empty())
        return res;

    size_t begin = 0;
    for (;;)
    {
        const size_t end = s.find(delimiter, begin);
        if (end == std::string::npos)
            break;

        res.emplace_back(s.substr(begin, end - begin));
        begin = end + delimiter.size();
    }

    res.emplace_back(s.substr(begin));

    return res;
}

void remove_whitespace(std::string& s);
bool is_whitespace(const std::string& s);

enum SyncCout {
    IO_LOCK,
    IO_UNLOCK
};
std::ostream& operator<<(std::ostream&, SyncCout);

#define sync_cout std::cout << IO_LOCK
#define sync_endl std::endl << IO_UNLOCK

void sync_cout_start();
void sync_cout_end();

// True if and only if the binary is compiled on a little-endian machine
static inline const union {
    uint32_t i;
    char     c[4];
} Le                                    = {0x01020304};
static inline const bool IsLittleEndian = (Le.c[0] == 4);

template<typename T, std::size_t MaxSize>
class ValueList {

   public:
    std::size_t size() const { return size_; }
    void        push_back(const T& value) { values_[size_++] = value; }
    const T*    begin() const { return values_; }
    const T*    end() const { return values_ + size_; }
    const T&    operator[](int index) const { return values_[index]; }

   private:
    T           values_[MaxSize];
    std::size_t size_ = 0;
};

// xorshift64star Pseudo-Random Number Generator
// This class is based on original code written and dedicated
// to the public domain by Sebastiano Vigna (2014).
// It has the following characteristics:
//
//  -  Outputs 64-bit numbers
//  -  Passes Dieharder and SmallCrush test batteries
//  -  Does not require warm-up, no zeroland to escape
//  -  Internal state is a single 64-bit integer
//  -  Period is 2^64 - 1
//  -  Speed: 1.60 ns/call (Core i7 @3.40GHz)
//
// For further analysis see
//   <http://vigna.di.unimi.it/ftp/papers/xorshift.pdf>

class PRNG {

    uint64_t s;

    uint64_t rand64() {

        s ^= s >> 12, s ^= s << 25, s ^= s >> 27;
        return s * 2685821657736338717LL;
    }

   public:
    PRNG(uint64_t seed) :
        s(seed) {
        assert(seed);
    }

    template<typename T>
    T rand() {
        return T(rand64());
    }

    // Special generator used to fast init magic numbers.
    // Output values only have 1/8th of their bits set on average.
    template<typename T>
    T sparse_rand() {
        return T(rand64() & rand64() & rand64());
    }
};

inline uint64_t mul_hi64(uint64_t a, uint64_t b) {
#if defined(__GNUC__) && defined(IS_64BIT)
    __extension__ using uint128 = unsigned __int128;
    return (uint128(a) * uint128(b)) >> 64;
#else
    uint64_t aL = uint32_t(a), aH = a >> 32;
    uint64_t bL = uint32_t(b), bH = b >> 32;
    uint64_t c1 = (aL * bL) >> 32;
    uint64_t c2 = aH * bL + c1;
    uint64_t c3 = aL * bH + uint32_t(c2);
    return aH * bH + (c2 >> 32) + (c3 >> 32);
#endif
}

struct CommandLine {
   public:
    CommandLine(int _argc, char** _argv) :
        argc(_argc),
        argv(_argv) {}

    static std::string get_binary_directory(std::string argv0);
    static std::string get_working_directory();

    int    argc;
    char** argv;
};

namespace Utility {

template<typename T, typename Predicate>
void move_to_front(std::vector<T>& vec, Predicate pred) {
    auto it = std::find_if(vec.begin(), vec.end(), pred);

    if (it != vec.end())
    {
        std::rotate(vec.begin(), it, it + 1);
    }
}
}

}  // namespace Stockfish

#endif  // #ifndef MISC_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef TYPES_H_INCLUDED
    #define TYPES_H_INCLUDED

// When compiling with provided Makefile (e.g. for Linux and OSX), configuration
// is done automatically. To get started type 'make help'.
//
// When Makefile is not used (e.g. with Microsoft Visual Studio) some switches
// need to be set manually:
//
// -DNDEBUG      | Disable debugging mode. Always use this for release.
//
// -DNO_PREFETCH | Disable use of prefetch asm-instruction. You may need this to
//               | run on some very old machines.
//
// -DUSE_POPCNT  | Add runtime support for use of popcnt asm-instruction. Works
//               | only in 64-bit mode and requires hardware with popcnt support.
//
// -DUSE_PEXT    | Add runtime support for use of pext asm-instruction. Works
//               | only in 64-bit mode and requires hardware with pext support.

    #if defined(_MSC_VER)
        // Disable some silly and noisy warnings from MSVC compiler
        #pragma warning(disable: 4127)  // Conditional expression is constant
        #pragma warning(disable: 4146)  // Unary minus operator applied to unsigned type
        #pragma warning(disable: 4800)  // Forcing value to bool 'true' or 'false'
    #endif

// Predefined macros hell:
//
// __GNUC__                Compiler is GCC, Clang or ICX
// __clang__               Compiler is Clang or ICX
// __INTEL_LLVM_COMPILER   Compiler is ICX
// _MSC_VER                Compiler is MSVC
// _WIN32                  Building on Windows (any)
// _WIN64                  Building on Windows 64 bit

    #if defined(__GNUC__) && (__GNUC__ < 9 || (__GNUC__ == 9 && __GNUC_MINOR__ <= 2)) \
      && defined(_WIN32) && !defined(__clang__)
        #define ALIGNAS_ON_STACK_VARIABLES_BROKEN
    #endif

    #define ASSERT_ALIGNED(ptr, alignment) assert(reinterpret_cast<uintptr_t>(ptr) % alignment == 0)

    #if defined(_WIN64) && defined(_MSC_VER)  // No Makefile used
        #define IS_64BIT
    #endif

    #if defined(USE_POPCNT) && defined(_MSC_VER)
    #endif

    #if !defined(NO_PREFETCH) && defined(_MSC_VER)
    #endif

    #if defined(USE_PEXT)
        #define pext(b, m) _pext_u64(b, m)
    #else
        #define pext(b, m) 0
    #endif

namespace Stockfish {

    #ifdef USE_POPCNT
constexpr bool HasPopCnt = true;
    #else
constexpr bool HasPopCnt = false;
    #endif

    #ifdef USE_PEXT
constexpr bool HasPext = true;
    #else
constexpr bool HasPext = false;
    #endif

    #ifdef IS_64BIT
constexpr bool Is64Bit = true;
    #else
constexpr bool Is64Bit = false;
    #endif

using Key      = uint64_t;
using Bitboard = uint64_t;

constexpr int MAX_MOVES = 256;
constexpr int MAX_PLY   = 246;

enum Color {
    WHITE,
    BLACK,
    COLOR_NB = 2
};

enum CastlingRights {
    NO_CASTLING,
    WHITE_OO,
    WHITE_OOO = WHITE_OO << 1,
    BLACK_OO  = WHITE_OO << 2,
    BLACK_OOO = WHITE_OO << 3,

    KING_SIDE      = WHITE_OO | BLACK_OO,
    QUEEN_SIDE     = WHITE_OOO | BLACK_OOO,
    WHITE_CASTLING = WHITE_OO | WHITE_OOO,
    BLACK_CASTLING = BLACK_OO | BLACK_OOO,
    ANY_CASTLING   = WHITE_CASTLING | BLACK_CASTLING,

    CASTLING_RIGHT_NB = 16
};

enum Bound {
    BOUND_NONE,
    BOUND_UPPER,
    BOUND_LOWER,
    BOUND_EXACT = BOUND_UPPER | BOUND_LOWER
};

// Value is used as an alias for int, this is done to differentiate between a search
// value and any other integer value. The values used in search are always supposed
// to be in the range (-VALUE_NONE, VALUE_NONE] and should not exceed this range.
using Value = int;

constexpr Value VALUE_ZERO     = 0;
constexpr Value VALUE_DRAW     = 0;
constexpr Value VALUE_NONE     = 32002;
constexpr Value VALUE_INFINITE = 32001;

constexpr Value VALUE_MATE             = 32000;
constexpr Value VALUE_MATE_IN_MAX_PLY  = VALUE_MATE - MAX_PLY;
constexpr Value VALUE_MATED_IN_MAX_PLY = -VALUE_MATE_IN_MAX_PLY;

constexpr Value VALUE_TB                 = VALUE_MATE_IN_MAX_PLY - 1;
constexpr Value VALUE_TB_WIN_IN_MAX_PLY  = VALUE_TB - MAX_PLY;
constexpr Value VALUE_TB_LOSS_IN_MAX_PLY = -VALUE_TB_WIN_IN_MAX_PLY;

// In the code, we make the assumption that these values
// are such that non_pawn_material() can be used to uniquely
// identify the material on the board.
constexpr Value PawnValue   = 208;
constexpr Value KnightValue = 781;
constexpr Value BishopValue = 825;
constexpr Value RookValue   = 1276;
constexpr Value QueenValue  = 2538;

// clang-format off
enum PieceType {
    NO_PIECE_TYPE, PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING,
    ALL_PIECES = 0,
    PIECE_TYPE_NB = 8
};

enum Piece {
    NO_PIECE,
    W_PAWN = PAWN,     W_KNIGHT, W_BISHOP, W_ROOK, W_QUEEN, W_KING,
    B_PAWN = PAWN + 8, B_KNIGHT, B_BISHOP, B_ROOK, B_QUEEN, B_KING,
    PIECE_NB = 16
};
// clang-format on

constexpr Value PieceValue[PIECE_NB] = {
  VALUE_ZERO, PawnValue, KnightValue, BishopValue, RookValue, QueenValue, VALUE_ZERO, VALUE_ZERO,
  VALUE_ZERO, PawnValue, KnightValue, BishopValue, RookValue, QueenValue, VALUE_ZERO, VALUE_ZERO};

using Depth = int;

enum : int {
    // The following DEPTH_ constants are used for transposition table entries
    // and quiescence search move generation stages. In regular search, the
    // depth stored in the transposition table is literal: the search depth
    // (effort) used to make the corresponding transposition table value. In
    // quiescence search, however, the transposition table entries only store
    // the current quiescence move generation stage (which should thus compare
    // lower than any regular search depth).
    DEPTH_QS = 0,
    // For transposition table entries where no searching at all was done
    // (whether regular or qsearch) we use DEPTH_UNSEARCHED, which should thus
    // compare lower than any quiescence or regular depth. DEPTH_ENTRY_OFFSET
    // is used only for the transposition table entry occupancy check (see tt.cpp),
    // and should thus be lower than DEPTH_UNSEARCHED.
    DEPTH_UNSEARCHED   = -2,
    DEPTH_ENTRY_OFFSET = -3
};

// clang-format off
enum Square : int {
    SQ_A1, SQ_B1, SQ_C1, SQ_D1, SQ_E1, SQ_F1, SQ_G1, SQ_H1,
    SQ_A2, SQ_B2, SQ_C2, SQ_D2, SQ_E2, SQ_F2, SQ_G2, SQ_H2,
    SQ_A3, SQ_B3, SQ_C3, SQ_D3, SQ_E3, SQ_F3, SQ_G3, SQ_H3,
    SQ_A4, SQ_B4, SQ_C4, SQ_D4, SQ_E4, SQ_F4, SQ_G4, SQ_H4,
    SQ_A5, SQ_B5, SQ_C5, SQ_D5, SQ_E5, SQ_F5, SQ_G5, SQ_H5,
    SQ_A6, SQ_B6, SQ_C6, SQ_D6, SQ_E6, SQ_F6, SQ_G6, SQ_H6,
    SQ_A7, SQ_B7, SQ_C7, SQ_D7, SQ_E7, SQ_F7, SQ_G7, SQ_H7,
    SQ_A8, SQ_B8, SQ_C8, SQ_D8, SQ_E8, SQ_F8, SQ_G8, SQ_H8,
    SQ_NONE,

    SQUARE_ZERO = 0,
    SQUARE_NB   = 64
};
// clang-format on

enum Direction : int {
    NORTH = 8,
    EAST  = 1,
    SOUTH = -NORTH,
    WEST  = -EAST,

    NORTH_EAST = NORTH + EAST,
    SOUTH_EAST = SOUTH + EAST,
    SOUTH_WEST = SOUTH + WEST,
    NORTH_WEST = NORTH + WEST
};

enum File : int {
    FILE_A,
    FILE_B,
    FILE_C,
    FILE_D,
    FILE_E,
    FILE_F,
    FILE_G,
    FILE_H,
    FILE_NB
};

enum Rank : int {
    RANK_1,
    RANK_2,
    RANK_3,
    RANK_4,
    RANK_5,
    RANK_6,
    RANK_7,
    RANK_8,
    RANK_NB
};

// Keep track of what a move changes on the board (used by NNUE)
struct DirtyPiece {

    // Number of changed pieces
    int dirty_num;

    // Max 3 pieces can change in one move. A promotion with capture moves
    // both the pawn and the captured piece to SQ_NONE and the piece promoted
    // to from SQ_NONE to the capture square.
    Piece piece[3];

    // From and to squares, which may be SQ_NONE
    Square from[3];
    Square to[3];
};

    #define ENABLE_INCR_OPERATORS_ON(T) \
        inline T& operator++(T& d) { return d = T(int(d) + 1); } \
        inline T& operator--(T& d) { return d = T(int(d) - 1); }

ENABLE_INCR_OPERATORS_ON(PieceType)
ENABLE_INCR_OPERATORS_ON(Square)
ENABLE_INCR_OPERATORS_ON(File)
ENABLE_INCR_OPERATORS_ON(Rank)

    #undef ENABLE_INCR_OPERATORS_ON

constexpr Direction operator+(Direction d1, Direction d2) { return Direction(int(d1) + int(d2)); }
constexpr Direction operator*(int i, Direction d) { return Direction(i * int(d)); }

// Additional operators to add a Direction to a Square
constexpr Square operator+(Square s, Direction d) { return Square(int(s) + int(d)); }
constexpr Square operator-(Square s, Direction d) { return Square(int(s) - int(d)); }
inline Square&   operator+=(Square& s, Direction d) { return s = s + d; }
inline Square&   operator-=(Square& s, Direction d) { return s = s - d; }

// Toggle color
constexpr Color operator~(Color c) { return Color(c ^ BLACK); }

// Swap A1 <-> A8
constexpr Square flip_rank(Square s) { return Square(s ^ SQ_A8); }

// Swap A1 <-> H1
constexpr Square flip_file(Square s) { return Square(s ^ SQ_H1); }

// Swap color of piece B_KNIGHT <-> W_KNIGHT
constexpr Piece operator~(Piece pc) { return Piece(pc ^ 8); }

constexpr CastlingRights operator&(Color c, CastlingRights cr) {
    return CastlingRights((c == WHITE ? WHITE_CASTLING : BLACK_CASTLING) & cr);
}

constexpr Value mate_in(int ply) { return VALUE_MATE - ply; }

constexpr Value mated_in(int ply) { return -VALUE_MATE + ply; }

constexpr Square make_square(File f, Rank r) { return Square((r << 3) + f); }

constexpr Piece make_piece(Color c, PieceType pt) { return Piece((c << 3) + pt); }

constexpr PieceType type_of(Piece pc) { return PieceType(pc & 7); }

inline Color color_of(Piece pc) {
    assert(pc != NO_PIECE);
    return Color(pc >> 3);
}

constexpr bool is_ok(Square s) { return s >= SQ_A1 && s <= SQ_H8; }

constexpr File file_of(Square s) { return File(s & 7); }

constexpr Rank rank_of(Square s) { return Rank(s >> 3); }

constexpr Square relative_square(Color c, Square s) { return Square(s ^ (c * 56)); }

constexpr Rank relative_rank(Color c, Rank r) { return Rank(r ^ (c * 7)); }

constexpr Rank relative_rank(Color c, Square s) { return relative_rank(c, rank_of(s)); }

constexpr Direction pawn_push(Color c) { return c == WHITE ? NORTH : SOUTH; }

// Based on a congruential pseudo-random number generator
constexpr Key make_key(uint64_t seed) {
    return seed * 6364136223846793005ULL + 1442695040888963407ULL;
}

enum MoveType {
    NORMAL,
    PROMOTION  = 1 << 14,
    EN_PASSANT = 2 << 14,
    CASTLING   = 3 << 14
};

// A move needs 16 bits to be stored
//
// bit  0- 5: destination square (from 0 to 63)
// bit  6-11: origin square (from 0 to 63)
// bit 12-13: promotion piece type - 2 (from KNIGHT-2 to QUEEN-2)
// bit 14-15: special move flag: promotion (1), en passant (2), castling (3)
// NOTE: en passant bit is set only when a pawn can be captured
//
// Special cases are Move::none() and Move::null(). We can sneak these in because
// in any normal move the destination square and origin square are always different,
// but Move::none() and Move::null() have the same origin and destination square.

class Move {
   public:
    Move() = default;
    constexpr explicit Move(std::uint16_t d) :
        data(d) {}

    constexpr Move(Square from, Square to) :
        data((from << 6) + to) {}

    template<MoveType T>
    static constexpr Move make(Square from, Square to, PieceType pt = KNIGHT) {
        return Move(T + ((pt - KNIGHT) << 12) + (from << 6) + to);
    }

    constexpr Square from_sq() const {
        assert(is_ok());
        return Square((data >> 6) & 0x3F);
    }

    constexpr Square to_sq() const {
        assert(is_ok());
        return Square(data & 0x3F);
    }

    constexpr int from_to() const { return data & 0xFFF; }

    constexpr MoveType type_of() const { return MoveType(data & (3 << 14)); }

    constexpr PieceType promotion_type() const { return PieceType(((data >> 12) & 3) + KNIGHT); }

    constexpr bool is_ok() const { return none().data != data && null().data != data; }

    static constexpr Move null() { return Move(65); }
    static constexpr Move none() { return Move(0); }

    constexpr bool operator==(const Move& m) const { return data == m.data; }
    constexpr bool operator!=(const Move& m) const { return data != m.data; }

    constexpr explicit operator bool() const { return data != 0; }

    constexpr std::uint16_t raw() const { return data; }

    struct MoveHash {
        std::size_t operator()(const Move& m) const { return make_key(m.data); }
    };

   protected:
    std::uint16_t data;
};

}  // namespace Stockfish

#endif  // #ifndef TYPES_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef MEMORY_H_INCLUDED
#define MEMORY_H_INCLUDED

namespace Stockfish {

void* std_aligned_alloc(size_t alignment, size_t size);
void  std_aligned_free(void* ptr);

// Memory aligned by page size, min alignment: 4096 bytes
void* aligned_large_pages_alloc(size_t size);
void  aligned_large_pages_free(void* mem);

// Frees memory which was placed there with placement new.
// Works for both single objects and arrays of unknown bound.
template<typename T, typename FREE_FUNC>
void memory_deleter(T* ptr, FREE_FUNC free_func) {
    if (!ptr)
        return;

    // Explicitly needed to call the destructor
    if constexpr (!std::is_trivially_destructible_v<T>)
        ptr->~T();

    free_func(ptr);
    return;
}

// Frees memory which was placed there with placement new.
// Works for both single objects and arrays of unknown bound.
template<typename T, typename FREE_FUNC>
void memory_deleter_array(T* ptr, FREE_FUNC free_func) {
    if (!ptr)
        return;

    // Move back on the pointer to where the size is allocated
    const size_t array_offset = std::max(sizeof(size_t), alignof(T));
    char*        raw_memory   = reinterpret_cast<char*>(ptr) - array_offset;

    if constexpr (!std::is_trivially_destructible_v<T>)
    {
        const size_t size = *reinterpret_cast<size_t*>(raw_memory);

        // Explicitly call the destructor for each element in reverse order
        for (size_t i = size; i-- > 0;)
            ptr[i].~T();
    }

    free_func(raw_memory);
}

// Allocates memory for a single object and places it there with placement new
template<typename T, typename ALLOC_FUNC, typename... Args>
inline std::enable_if_t<!std::is_array_v<T>, T*> memory_allocator(ALLOC_FUNC alloc_func,
                                                                   Args&&... args) {
    void* raw_memory = alloc_func(sizeof(T));
    ASSERT_ALIGNED(raw_memory, alignof(T));
    return new (raw_memory) T(std::forward<Args>(args)...);
}

// Allocates memory for an array of unknown bound and places it there with placement new
template<typename T, typename ALLOC_FUNC>
inline std::enable_if_t<std::is_array_v<T>, std::remove_extent_t<T>*>
memory_allocator(ALLOC_FUNC alloc_func, size_t num) {
    using ElementType = std::remove_extent_t<T>;

    const size_t array_offset = std::max(sizeof(size_t), alignof(ElementType));

    // Save the array size in the memory location
    char* raw_memory =
      reinterpret_cast<char*>(alloc_func(array_offset + num * sizeof(ElementType)));
    ASSERT_ALIGNED(raw_memory, alignof(T));

    new (raw_memory) size_t(num);

    for (size_t i = 0; i < num; ++i)
        new (raw_memory + array_offset + i * sizeof(ElementType)) ElementType();

    // Need to return the pointer at the start of the array so that
    // the indexing in unique_ptr<T[]> works.
    return reinterpret_cast<ElementType*>(raw_memory + array_offset);
}

//
//
// aligned large page unique ptr
//
//

template<typename T>
struct LargePageDeleter {
    void operator()(T* ptr) const { return memory_deleter<T>(ptr, aligned_large_pages_free); }
};

template<typename T>
struct LargePageArrayDeleter {
    void operator()(T* ptr) const { return memory_deleter_array<T>(ptr, aligned_large_pages_free); }
};

template<typename T>
using LargePagePtr =
  std::conditional_t<std::is_array_v<T>,
                     std::unique_ptr<T, LargePageArrayDeleter<std::remove_extent_t<T>>>,
                     std::unique_ptr<T, LargePageDeleter<T>>>;

// make_unique_large_page for single objects
```

```
template<typename T, typename... Args>
std::enable_if_t<!std::is_array_v<T>, LargePagePtr<T>> make_unique_large_page(Args&&... args) {
    static_assert(alignof(T) <= 4096,
                  "aligned_large_pages_alloc() may fail for such a big alignment requirement of T");

    T* obj = memory_allocator<T>(aligned_large_pages_alloc, std::forward<Args>(args)...);

    return LargePagePtr<T>(obj);
}

// make_unique_large_page for arrays of unknown bound
template<typename T>
std::enable_if_t<std::is_array_v<T>, LargePagePtr<T>> make_unique_large_page(size_t num) {
    using ElementType = std::remove_extent_t<T>;

    static_assert(alignof(ElementType) <= 4096,
                  "aligned_large_pages_alloc() may fail for such a big alignment requirement of T");

    ElementType* memory = memory_allocator<T>(aligned_large_pages_alloc, num);

    return LargePagePtr<T>(memory);
}

//
//
// aligned unique ptr
//
//

template<typename T>
struct AlignedDeleter {
    void operator()(T* ptr) const { return memory_deleter<T>(ptr, std_aligned_free); }
};

template<typename T>
struct AlignedArrayDeleter {
    void operator()(T* ptr) const { return memory_deleter_array<T>(ptr, std_aligned_free); }
};

template<typename T>
using AlignedPtr =
  std::conditional_t<std::is_array_v<T>,
                     std::unique_ptr<T, AlignedArrayDeleter<std::remove_extent_t<T>>>,
                     std::unique_ptr<T, AlignedDeleter<T>>>;

// make_unique_aligned for single objects
template<typename T, typename... Args>
std::enable_if_t<!std::is_array_v<T>, AlignedPtr<T>> make_unique_aligned(Args&&... args) {
    const auto func = [](size_t size) { return std_aligned_alloc(alignof(T), size); };
    T*         obj  = memory_allocator<T>(func, std::forward<Args>(args)...);

    return AlignedPtr<T>(obj);
}

// make_unique_aligned for arrays of unknown bound
template<typename T>
std::enable_if_t<std::is_array_v<T>, AlignedPtr<T>> make_unique_aligned(size_t num) {
    using ElementType = std::remove_extent_t<T>;

    const auto   func   = [](size_t size) { return std_aligned_alloc(alignof(ElementType), size); };
    ElementType* memory = memory_allocator<T>(func, num);

    return AlignedPtr<T>(memory);
}

// Get the first aligned element of an array.
// ptr must point to an array of size at least `sizeof(T) * N + alignment` bytes,
// where N is the number of elements in the array.
template<uintptr_t Alignment, typename T>
T* align_ptr_up(T* ptr) {
    static_assert(alignof(T) < Alignment);

    const uintptr_t ptrint = reinterpret_cast<uintptr_t>(reinterpret_cast<char*>(ptr));
    return reinterpret_cast<T*>(
      reinterpret_cast<char*>((ptrint + (Alignment - 1)) / Alignment * Alignment));
}

}  // namespace Stockfish

#endif  // #ifndef MEMORY_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef NUMA_H_INCLUDED
#define NUMA_H_INCLUDED

// We support linux very well, but we explicitly do NOT support Android,
// because there is no affected systems, not worth maintaining.
#if defined(__linux__) && !defined(__ANDROID__)
    #if !defined(_GNU_SOURCE)
        #define _GNU_SOURCE
    #endif
#elif defined(_WIN64)

    #if _WIN32_WINNT < 0x0601
        #undef _WIN32_WINNT
        #define _WIN32_WINNT 0x0601  // Force to include needed API prototypes
    #endif

// On Windows each processor group can have up to 64 processors.
// https://learn.microsoft.com/en-us/windows/win32/procthread/processor-groups
static constexpr size_t WIN_PROCESSOR_GROUP_SIZE = 64;

    #if !defined(NOMINMAX)
        #define NOMINMAX
    #endif
    #if defined small
        #undef small
    #endif

// https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadselectedcpusetmasks
using SetThreadSelectedCpuSetMasks_t = BOOL (*)(HANDLE, PGROUP_AFFINITY, USHORT);

// https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getthreadselectedcpusetmasks
using GetThreadSelectedCpuSetMasks_t = BOOL (*)(HANDLE, PGROUP_AFFINITY, USHORT, PUSHORT);

#endif

namespace Stockfish {

using CpuIndex  = size_t;
using NumaIndex = size_t;

inline CpuIndex get_hardware_concurrency() {
    CpuIndex concurrency = std::thread::hardware_concurrency();

    // Get all processors across all processor groups on windows, since
    // hardware_concurrency() only returns the number of processors in
    // the first group, because only these are available to std::thread.
#ifdef _WIN64
    concurrency = std::max<CpuIndex>(concurrency, GetActiveProcessorCount(ALL_PROCESSOR_GROUPS));
#endif

    return concurrency;
}

inline const CpuIndex SYSTEM_THREADS_NB = std::max<CpuIndex>(1, get_hardware_concurrency());

#if defined(_WIN64)

struct WindowsAffinity {
    std::optional<std::set<CpuIndex>> oldApi;
    std::optional<std::set<CpuIndex>> newApi;

    // We also provide diagnostic for when the affinity is set to nullopt
    // whether it was due to being indeterminate. If affinity is indeterminate
    // it is best to assume it is not set at all, so consistent with the meaning
    // of the nullopt affinity.
    bool isNewDeterminate = true;
    bool isOldDeterminate = true;

    std::optional<std::set<CpuIndex>> get_combined() const {
        if (!oldApi.has_value())
            return newApi;
        if (!newApi.has_value())
            return oldApi;

        std::set<CpuIndex> intersect;
        std::set_intersection(oldApi->begin(), oldApi->end(), newApi->begin(), newApi->end(),
                              std::inserter(intersect, intersect.begin()));
        return intersect;
    }

    // Since Windows 11 and Windows Server 2022 thread affinities can span
    // processor groups and can be set as such by a new WinAPI function. However,
    // we may need to force using the old API if we detect that the process has
    // affinity set by the old API already and we want to override that. Due to the
    // limitations of the old API we cannot detect its use reliably. There will be
    // cases where we detect not use but it has actually been used and vice versa.

    bool likely_used_old_api() const { return oldApi.has_value() || !isOldDeterminate; }
};

inline std::pair<BOOL, std::vector<USHORT>> get_process_group_affinity() {

    // GetProcessGroupAffinity requires the GroupArray argument to be
    // aligned to 4 bytes instead of just 2.
    static constexpr size_t GroupArrayMinimumAlignment = 4;
    static_assert(GroupArrayMinimumAlignment >= alignof(USHORT));

    // The function should succeed the second time, but it may fail if the group
    // affinity has changed between GetProcessGroupAffinity calls. In such case
    // we consider this a hard error, as we Cannot work with unstable affinities
    // anyway.
    static constexpr int MAX_TRIES  = 2;
    USHORT               GroupCount = 1;
    for (int i = 0; i < MAX_TRIES; ++i)
    {
        auto GroupArray = std::make_unique<USHORT[]>(
          GroupCount + (GroupArrayMinimumAlignment / alignof(USHORT) - 1));

        USHORT* GroupArrayAligned = align_ptr_up<GroupArrayMinimumAlignment>(GroupArray.get());

        const BOOL status =
          GetProcessGroupAffinity(GetCurrentProcess(), &GroupCount, GroupArrayAligned);

        if (status == 0 && GetLastError() != ERROR_INSUFFICIENT_BUFFER)
        {
            break;
        }

        if (status != 0)
        {
            return std::make_pair(status,
                                  std::vector(GroupArrayAligned, GroupArrayAligned + GroupCount));
        }
    }

    return std::make_pair(0, std::vector<USHORT>());
}

// On Windows there are two ways to set affinity, and therefore 2 ways to get it.
// These are not consistent, so we have to check both. In some cases it is actually
// not possible to determine affinity. For example when two different threads have
// affinity on different processor groups, set using SetThreadAffinityMask, we cannot
// retrieve the actual affinities.
// From documentation on GetProcessAffinityMask:
//     > If the calling process contains threads in multiple groups,
//     > the function returns zero for both affinity masks.
// In such cases we just give up and assume we have affinity for all processors.
// nullopt means no affinity is set, that is, all processors are allowed
inline WindowsAffinity get_process_affinity() {
    HMODULE k32                            = GetModuleHandle(TEXT("Kernel32.dll"));
    auto    GetThreadSelectedCpuSetMasks_f = GetThreadSelectedCpuSetMasks_t(
      (void (*)()) GetProcAddress(k32, "GetThreadSelectedCpuSetMasks"));

    BOOL status = 0;

    WindowsAffinity affinity;

    if (GetThreadSelectedCpuSetMasks_f != nullptr)
    {
        USHORT RequiredMaskCount;
        status = GetThreadSelectedCpuSetMasks_f(GetCurrentThread(), nullptr, 0, &RequiredMaskCount);

        // We expect ERROR_INSUFFICIENT_BUFFER from GetThreadSelectedCpuSetMasks,
        // but other failure is an actual error.
        if (status == 0 && GetLastError() != ERROR_INSUFFICIENT_BUFFER)
        {
            affinity.isNewDeterminate = false;
        }
        else if (RequiredMaskCount > 0)
        {
            // If RequiredMaskCount then these affinities were never set, but it's
            // not consistent so GetProcessAffinityMask may still return some affinity.
            auto groupAffinities = std::make_unique<GROUP_AFFINITY[]>(RequiredMaskCount);

            status = GetThreadSelectedCpuSetMasks_f(GetCurrentThread(), groupAffinities.get(),
                                                    RequiredMaskCount, &RequiredMaskCount);

            if (status == 0)
            {
                affinity.isNewDeterminate = false;
            }
            else
            {
                std::set<CpuIndex> cpus;

                for (USHORT i = 0; i < RequiredMaskCount; ++i)
                {
                    const size_t procGroupIndex = groupAffinities[i].Group;

                    for (size_t j = 0; j < WIN_PROCESSOR_GROUP_SIZE; ++j)
                    {
                        if (groupAffinities[i].Mask & (KAFFINITY(1) << j))
                            cpus.insert(procGroupIndex * WIN_PROCESSOR_GROUP_SIZE + j);
                    }
                }

                affinity.newApi = std::move(cpus);
            }
        }
    }

    // NOTE: There is no way to determine full affinity using the old API if
    //       individual threads set affinity on different processor groups.

    DWORD_PTR proc, sys;
    status = GetProcessAffinityMask(GetCurrentProcess(), &proc, &sys);

    // If proc == 0 then we cannot determine affinity because it spans processor groups.
    // On Windows 11 and Server 2022 it will instead
    //     > If, however, hHandle specifies a handle to the current process, the function
    //     > always uses the calling thread's primary group (which by default is the same
    //     > as the process' primary group) in order to set the
    //     > lpProcessAffinityMask and lpSystemAffinityMask.
    // So it will never be indeterminate here. We can only make assumptions later.
    if (status == 0 || proc == 0)
    {
        affinity.isOldDeterminate = false;
        return affinity;
    }

    // If SetProcessAffinityMask was never called the affinity must span
    // all processor groups, but if it was called it must only span one.

    std::vector<USHORT> groupAffinity;  // We need to capture this later and capturing
                                        // from structured bindings requires c++20.

    std::tie(status, groupAffinity) = get_process_group_affinity();
    if (status == 0)
    {
        affinity.isOldDeterminate = false;
        return affinity;
    }

    if (groupAffinity.size() == 1)
    {
        // We detect the case when affinity is set to all processors and correctly
        // leave affinity.oldApi as nullopt.
        if (GetActiveProcessorGroupCount() != 1 || proc != sys)
        {
            std::set<CpuIndex> cpus;

            const size_t procGroupIndex = groupAffinity[0];

            const uint64_t mask = static_cast<uint64_t>(proc);
            for (size_t j = 0; j < WIN_PROCESSOR_GROUP_SIZE; ++j)
            {
                if (mask & (KAFFINITY(1) << j))
                    cpus.insert(procGroupIndex * WIN_PROCESSOR_GROUP_SIZE + j);
            }

            affinity.oldApi = std::move(cpus);
        }
    }
    else
    {
        // If we got here it means that either SetProcessAffinityMask was never set
        // or we're on Windows 11/Server 2022.

        // Since Windows 11 and Windows Server 2022 the behaviour of
        // GetProcessAffinityMask changed:
        //     > If, however, hHandle specifies a handle to the current process,
        //     > the function always uses the calling thread's primary group
        //     > (which by default is the same as the process' primary group)
        //     > in order to set the lpProcessAffinityMask and lpSystemAffinityMask.
        // In which case we can actually retrieve the full affinity.

        if (GetThreadSelectedCpuSetMasks_f != nullptr)
        {
            std::thread th([&]() {
                std::set<CpuIndex> cpus;
                bool               isAffinityFull = true;

                for (auto procGroupIndex : groupAffinity)
                {
                    const int numActiveProcessors =
                      GetActiveProcessorCount(static_cast<WORD>(procGroupIndex));

                    // We have to schedule to two different processors
                    // and & the affinities we get. Otherwise our processor
                    // choice could influence the resulting affinity.
                    // We assume the processor IDs within the group are
                    // filled sequentially from 0.
                    uint64_t procCombined = std::numeric_limits<uint64_t>::max();
                    uint64_t sysCombined  = std::numeric_limits<uint64_t>::max();

                    for (int i = 0; i < std::min(numActiveProcessors, 2); ++i)
                    {
                        GROUP_AFFINITY GroupAffinity;
                        std::memset(&GroupAffinity, 0, sizeof(GROUP_AFFINITY));
                        GroupAffinity.Group = static_cast<WORD>(procGroupIndex);

                        GroupAffinity.Mask = static_cast<KAFFINITY>(1) << i;

                        status =
                          SetThreadGroupAffinity(GetCurrentThread(), &GroupAffinity, nullptr);
                        if (status == 0)
                        {
                            affinity.isOldDeterminate = false;
                            return;
                        }

                        SwitchToThread();

                        DWORD_PTR proc2, sys2;
                        status = GetProcessAffinityMask(GetCurrentProcess(), &proc2, &sys2);
                        if (status == 0)
                        {
                            affinity.isOldDeterminate = false;
                            return;
                        }

                        procCombined &= static_cast<uint64_t>(proc2);
                        sysCombined &= static_cast<uint64_t>(sys2);
                    }

                    if (procCombined != sysCombined)
                        isAffinityFull = false;

                    for (size_t j = 0; j < WIN_PROCESSOR_GROUP_SIZE; ++j)
                    {
                        if (procCombined & (KAFFINITY(1) << j))
                            cpus.insert(procGroupIndex * WIN_PROCESSOR_GROUP_SIZE + j);
                    }
                }

                // We have to detect the case where the affinity was not set,
                // or is set to all processors so that we correctly produce as
                // std::nullopt result.
                if (!isAffinityFull)
                {
                    affinity.oldApi = std::move(cpus);
                }
            });

            th.join();
        }
    }

    return affinity;
}

#endif

#if defined(__linux__) && !defined(__ANDROID__)

inline std::set<CpuIndex> get_process_affinity() {

    std::set<CpuIndex> cpus;

    // For unsupported systems, or in case of a soft error, we may assume
    // all processors are available for use.
    [[maybe_unused]] auto set_to_all_cpus = [&]() {
        for (CpuIndex c = 0; c < SYSTEM_THREADS_NB; ++c)
            cpus.insert(c);
    };

    // cpu_set_t by default holds 1024 entries. This may not be enough soon,
    // but there is no easy way to determine how many threads there actually
    // is. In this case we just choose a reasonable upper bound.
    static constexpr CpuIndex MaxNumCpus = 1024 * 64;

    cpu_set_t* mask = CPU_ALLOC(MaxNumCpus);
    if (mask == nullptr)
        std::exit(EXIT_FAILURE);

    const size_t masksize = CPU_ALLOC_SIZE(MaxNumCpus);

    CPU_ZERO_S(masksize, mask);

    const int status = sched_getaffinity(0, masksize, mask);

    if (status != 0)
    {
        CPU_FREE(mask);
        std::exit(EXIT_FAILURE);
    }

    for (CpuIndex c = 0; c < MaxNumCpus; ++c)
        if (CPU_ISSET_S(c, masksize, mask))
            cpus.insert(c);

    CPU_FREE(mask);

    return cpus;
}

#endif

#if defined(__linux__) && !defined(__ANDROID__)

inline static const auto STARTUP_PROCESSOR_AFFINITY = get_process_affinity();

#elif defined(_WIN64)

inline static const auto STARTUP_PROCESSOR_AFFINITY = get_process_affinity();
inline static const auto STARTUP_USE_OLD_AFFINITY_API =
  STARTUP_PROCESSOR_AFFINITY.likely_used_old_api();

#endif

// We want to abstract the purpose of storing the numa node index somewhat.
// Whoever is using this does not need to know the specifics of the replication
// machinery to be able to access NUMA replicated memory.
class NumaReplicatedAccessToken {
   public:
    NumaReplicatedAccessToken() :
        n(0) {}

    explicit NumaReplicatedAccessToken(NumaIndex idx) :
        n(idx) {}

    NumaIndex get_numa_index() const { return n; }

   private:
    NumaIndex n;
};

// Designed as immutable, because there is no good reason to alter an already
// existing config in a way that doesn't require recreating it completely, and
// it would be complex and expensive to maintain class invariants.
// The CPU (processor) numbers always correspond to the actual numbering used
// by the system. The NUMA node numbers MAY NOT correspond to the system's
// numbering of the NUMA nodes. In particular, empty nodes may be removed, or
// the user may create custom nodes. It is guaranteed that NUMA nodes are NOT
// empty: every node exposed by NumaConfig has at least one processor assigned.
//
// We use startup affinities so as not to modify its own behaviour in time.
//
// Since Stockfish doesn't support exceptions all places where an exception
// should be thrown are replaced by std::exit.
class NumaConfig {
   public:
    NumaConfig() :
        highestCpuIndex(0),
        customAffinity(false) {
        const auto numCpus = SYSTEM_THREADS_NB;
        add_cpu_range_to_node(NumaIndex{0}, CpuIndex{0}, numCpus - 1);
    }

    // This function queries the system for the mapping of processors to NUMA nodes.
    // On Linux we read from standardized kernel sysfs, with a fallback to single NUMA
    // node. On Windows we utilize GetNumaProcessorNodeEx, which has its quirks, see
    // comment for Windows implementation of get_process_affinity.
    static NumaConfig from_system([[maybe_unused]] bool respectProcessAffinity = true) {
        NumaConfig cfg = empty();

#if defined(__linux__) && !defined(__ANDROID__)

        std::set<CpuIndex> allowedCpus;

        if (respectProcessAffinity)
            allowedCpus = STARTUP_PROCESSOR_AFFINITY;

        auto is_cpu_allowed = [respectProcessAffinity, &allowedCpus](CpuIndex c) {
            return !respectProcessAffinity || allowedCpus.count(c) == 1;
        };

        // On Linux things are straightforward, since there's no processor groups and
        // any thread can be scheduled on all processors.
        // We try to gather this information from the sysfs first
        // https://www.kernel.org/doc/Documentation/ABI/stable/sysfs-devices-node

        bool useFallback = false;
        auto fallback    = [&]() {
            useFallback = true;
            cfg         = empty();
        };

        // /sys/devices/system/node/online contains information about active NUMA nodes
        auto nodeIdsStr = read_file_to_string("/sys/devices/system/node/online");
        if (!nodeIdsStr.has_value() || nodeIdsStr->empty())
        {
            fallback();
        }
        else
        {
            remove_whitespace(*nodeIdsStr);
            for (size_t n : indices_from_shortened_string(*nodeIdsStr))
            {
                // /sys/devices/system/node/node.../cpulist
                std::string path =
                  std::string("/sys/devices/system/node/node") + std::to_string(n) + "/cpulist";
                auto cpuIdsStr = read_file_to_string(path);
                // Now, we only bail if the file does not exist. Some nodes may be
                // empty, that's fine. An empty node still has a file that appears
                // to have some whitespace, so we need to handle that.
                if (!cpuIdsStr.has_value())
                {
                    fallback();
                    break;
                }
                else
                {
                    remove_whitespace(*cpuIdsStr);
                    for (size_t c : indices_from_shortened_string(*cpuIdsStr))
                    {
                        if (is_cpu_allowed(c))
                            cfg.add_cpu_to_node(n, c);
                    }
                }
            }
        }

        if (useFallback)
        {
            for (CpuIndex c = 0; c < SYSTEM_THREADS_NB; ++c)
                if (is_cpu_allowed(c))
                    cfg.add_cpu_to_node(NumaIndex{0}, c);
        }

#elif defined(_WIN64)

        std::optional<std::set<CpuIndex>> allowedCpus;

        if (respectProcessAffinity)
            allowedCpus = STARTUP_PROCESSOR_AFFINITY.get_combined();

        // The affinity cannot be determined in all cases on Windows,
        // but we at least guarantee that the number of allowed processors
        // is >= number of processors in the affinity mask. In case the user
        // is not satisfied they must set the processor numbers explicitly.
        auto is_cpu_allowed = [&allowedCpus](CpuIndex c) {
            return !allowedCpus.has_value() || allowedCpus->count(c) == 1;
        };

        WORD numProcGroups = GetActiveProcessorGroupCount();
        for (WORD procGroup = 0; procGroup < numProcGroups; ++procGroup)
        {
            for (BYTE number = 0; number < WIN_PROCESSOR_GROUP_SIZE; ++number)
            {
                PROCESSOR_NUMBER procnum;
                procnum.Group    = procGroup;
                procnum.Number   = number;
                procnum.Reserved = 0;
                USHORT nodeNumber;

                const BOOL     status = GetNumaProcessorNodeEx(&procnum, &nodeNumber);
                const CpuIndex c      = static_cast<CpuIndex>(procGroup) * WIN_PROCESSOR_GROUP_SIZE
                                 + static_cast<CpuIndex>(number);
                if (status != 0 && nodeNumber != std::numeric_limits<USHORT>::max()
                    && is_cpu_allowed(c))
                {
                    cfg.add_cpu_to_node(nodeNumber, c);
                }
            }
        }

        // Split the NUMA nodes to be contained within a group if necessary.
        // This is needed between Windows 10 Build 20348 and Windows 11, because
        // the new NUMA allocation behaviour was introduced while there was
        // still no way to set thread affinity spanning multiple processor groups.
        // See https://learn.microsoft.com/en-us/windows/win32/procthread/numa-support
        // We also do this is if need to force old API for some reason.
        //
        // 2024-08-26: It appears that we need to actually always force this behaviour.
        // While Windows allows this to work now, such assignments have bad interaction
        // with the scheduler - in particular it still prefers scheduling on the thread's
        // "primary" node, even if it means scheduling SMT processors first.
        // See https://github.com/official-stockfish/Stockfish/issues/5551
        // See https://learn.microsoft.com/en-us/windows/win32/procthread/processor-groups
        //
        //     Each process is assigned a primary group at creation, and by default all
        //     of its threads' primary group is the same. Each thread's ideal processor
        //     is in the thread's primary group, so threads will preferentially be
        //     scheduled to processors on their primary group, but they are able to
        //     be scheduled to processors on any other group.
        //
        // used to be guarded by if (STARTUP_USE_OLD_AFFINITY_API)
        {
            NumaConfig splitCfg = empty();

            NumaIndex splitNodeIndex = 0;
            for (const auto& cpus : cfg.nodes)
            {
                if (cpus.empty())
                    continue;

                size_t lastProcGroupIndex = *(cpus.begin()) / WIN_PROCESSOR_GROUP_SIZE;
                for (CpuIndex c : cpus)
                {
                    const size_t procGroupIndex = c / WIN_PROCESSOR_GROUP_SIZE;
                    if (procGroupIndex != lastProcGroupIndex)
                    {
                        splitNodeIndex += 1;
                        lastProcGroupIndex = procGroupIndex;
                    }
                    splitCfg.add_cpu_to_node(splitNodeIndex, c);
                }
                splitNodeIndex += 1;
            }

            cfg = std::move(splitCfg);
        }

#else

        // Fallback for unsupported systems.
        for (CpuIndex c = 0; c < SYSTEM_THREADS_NB; ++c)
            cfg.add_cpu_to_node(NumaIndex{0}, c);

#endif

        // We have to ensure no empty NUMA nodes persist.
        cfg.remove_empty_numa_nodes();

        // If the user explicitly opts out from respecting the current process affinity
        // then it may be inconsistent with the current affinity (obviously), so we
        // consider it custom.
        if (!respectProcessAffinity)
            cfg.customAffinity = true;

        return cfg;
    }

    // ':'-separated numa nodes
    // ','-separated cpu indices
    // supports "first-last" range syntax for cpu indices
    // For example "0-15,128-143:16-31,144-159:32-47,160-175:48-63,176-191"
    static NumaConfig from_string(const std::string& s) {
        NumaConfig cfg = empty();

        NumaIndex n = 0;
        for (auto&& nodeStr : split(s, ":"))
        {
            auto indices = indices_from_shortened_string(nodeStr);
            if (!indices.empty())
            {
                for (auto idx : indices)
                {
                    if (!cfg.add_cpu_to_node(n, CpuIndex(idx)))
                        std::exit(EXIT_FAILURE);
                }

                n += 1;
            }
        }

        cfg.customAffinity = true;

        return cfg;
    }

    NumaConfig(const NumaConfig&)            = delete;
    NumaConfig(NumaConfig&&)                 = default;
    NumaConfig& operator=(const NumaConfig&) = delete;
    NumaConfig& operator=(NumaConfig&&)      = default;

    bool is_cpu_assigned(CpuIndex n) const { return nodeByCpu.count(n) == 1; }

    NumaIndex num_numa_nodes() const { return nodes.size(); }

    CpuIndex num_cpus_in_numa_node(NumaIndex n) const {
        assert(n < nodes.size());
        return nodes[n].size();
    }

    CpuIndex num_cpus() const { return nodeByCpu.size(); }

    bool requires_memory_replication() const { return customAffinity || nodes.size() > 1; }

    std::string to_string() const {
        std::string str;

        bool isFirstNode = true;
        for (auto&& cpus : nodes)
        {
            if (!isFirstNode)
                str += ":";

            bool isFirstSet = true;
            auto rangeStart = cpus.begin();
            for (auto it = cpus.begin(); it != cpus.end(); ++it)
            {
                auto next = std::next(it);
                if (next == cpus.end() || *next != *it + 1)
                {
                    // cpus[i] is at the end of the range (may be of size 1)
                    if (!isFirstSet)
                        str += ",";

                    const CpuIndex last = *it;

                    if (it != rangeStart)
                    {
                        const CpuIndex first = *rangeStart;

                        str += std::to_string(first);
                        str += "-";
                        str += std::to_string(last);
                    }
                    else
                        str += std::to_string(last);

                    rangeStart = next;
                    isFirstSet = false;
                }
            }

            isFirstNode = false;
        }

        return str;
    }

    bool suggests_binding_threads(CpuIndex numThreads) const {
        // If we can reasonably determine that the threads cannot be contained
        // by the OS within the first NUMA node then we advise distributing
        // and binding threads. When the threads are not bound we can only use
        // NUMA memory replicated objects from the first node, so when the OS
        // has to schedule on other nodes we lose performance. We also suggest
        // binding if there's enough threads to distribute among nodes with minimal
        // disparity. We try to ignore small nodes, in particular the empty ones.

        // If the affinity set by the user does not match the affinity given by
        // the OS then binding is necessary to ensure the threads are running on
        // correct processors.
        if (customAffinity)
            return true;

        // We obviously cannot distribute a single thread, so a single thread
        // should never be bound.
        if (numThreads <= 1)
            return false;

        size_t largestNodeSize = 0;
        for (auto&& cpus : nodes)
            if (cpus.size() > largestNodeSize)
                largestNodeSize = cpus.size();

        auto is_node_small = [largestNodeSize](const std::set<CpuIndex>& node) {
            static constexpr double SmallNodeThreshold = 0.6;
            return static_cast<double>(node.size()) / static_cast<double>(largestNodeSize)
                <= SmallNodeThreshold;
        };

        size_t numNotSmallNodes = 0;
        for (auto&& cpus : nodes)
            if (!is_node_small(cpus))
                numNotSmallNodes += 1;

        return (numThreads > largestNodeSize / 2 || numThreads >= numNotSmallNodes * 4)
            && nodes.size() > 1;
    }

    std::vector<NumaIndex> distribute_threads_among_numa_nodes(CpuIndex numThreads) const {
        std::vector<NumaIndex> ns;

        if (nodes.size() == 1)
        {
            // Special case for when there's no NUMA nodes. This doesn't buy us
            // much, but let's keep the default path simple.
            ns.resize(numThreads, NumaIndex{0});
        }
        else
        {
            std::vector<size_t> occupation(nodes.size(), 0);
            for (CpuIndex c = 0; c < numThreads; ++c)
            {
                NumaIndex bestNode{0};
                float     bestNodeFill = std::numeric_limits<float>::max();
                for (NumaIndex n = 0; n < nodes.size(); ++n)
                {
                    float fill =
                      static_cast<float>(occupation[n] + 1) / static_cast<float>(nodes[n].size());
                    // NOTE: Do we want to perhaps fill the first available node
                    //       up to 50% first before considering other nodes?
                    //       Probably not, because it would interfere with running
                    //       multiple instances. We basically shouldn't favor any
                    //       particular node.
                    if (fill < bestNodeFill)
                    {
                        bestNode     = n;
                        bestNodeFill = fill;
                    }
                }
                ns.emplace_back(bestNode);
                occupation[bestNode] += 1;
            }
        }

        return ns;
    }

    NumaReplicatedAccessToken bind_current_thread_to_numa_node(NumaIndex n) const {
        if (n >= nodes.size() || nodes[n].size() == 0)
            std::exit(EXIT_FAILURE);

#if defined(__linux__) && !defined(__ANDROID__)

        cpu_set_t* mask = CPU_ALLOC(highestCpuIndex + 1);
        if (mask == nullptr)
            std::exit(EXIT_FAILURE);

        const size_t masksize = CPU_ALLOC_SIZE(highestCpuIndex + 1);

        CPU_ZERO_S(masksize, mask);

        for (CpuIndex c : nodes[n])
            CPU_SET_S(c, masksize, mask);

        const int status = sched_setaffinity(0, masksize, mask);

        CPU_FREE(mask);

        if (status != 0)
            std::exit(EXIT_FAILURE);

        // We yield this thread just to be sure it gets rescheduled.
        // This is defensive, allowed because this code is not performance critical.
        sched_yield();

#elif defined(_WIN64)

        // Requires Windows 11. No good way to set thread affinity spanning
        // processor groups before that.
        HMODULE k32                            = GetModuleHandle(TEXT("Kernel32.dll"));
        auto    SetThreadSelectedCpuSetMasks_f = SetThreadSelectedCpuSetMasks_t(
          (void (*)()) GetProcAddress(k32, "SetThreadSelectedCpuSetMasks"));

        // We ALWAYS set affinity with the new API if available, because
        // there's no downsides, and we forcibly keep it consistent with
        // the old API should we need to use it. I.e. we always keep this
        // as a superset of what we set with SetThreadGroupAffinity.
        if (SetThreadSelectedCpuSetMasks_f != nullptr)
        {
            // Only available on Windows 11 and Windows Server 2022 onwards
            const USHORT numProcGroups = USHORT(
              ((highestCpuIndex + 1) + WIN_PROCESSOR_GROUP_SIZE - 1) / WIN_PROCESSOR_GROUP_SIZE);
            auto groupAffinities = std::make_unique<GROUP_AFFINITY[]>(numProcGroups);
            std::memset(groupAffinities.get(), 0, sizeof(GROUP_AFFINITY) * numProcGroups);
            for (WORD i = 0; i < numProcGroups; ++i)
                groupAffinities[i].Group = i;

            for (CpuIndex c : nodes[n])
            {
                const size_t procGroupIndex     = c / WIN_PROCESSOR_GROUP_SIZE;
                const size_t idxWithinProcGroup = c % WIN_PROCESSOR_GROUP_SIZE;
                groupAffinities[procGroupIndex].Mask |= KAFFINITY(1) << idxWithinProcGroup;
            }

            HANDLE hThread = GetCurrentThread();

            const BOOL status =
              SetThreadSelectedCpuSetMasks_f(hThread, groupAffinities.get(), numProcGroups);
            if (status == 0)
                std::exit(EXIT_FAILURE);

            // We yield this thread just to be sure it gets rescheduled.
            // This is defensive, allowed because this code is not performance critical.
            SwitchToThread();
        }

        // Sometimes we need to force the old API, but do not use it unless necessary.
        if (SetThreadSelectedCpuSetMasks_f == nullptr || STARTUP_USE_OLD_AFFINITY_API)
        {
            // On earlier windows version (since windows 7) we cannot run a single thread
            // on multiple processor groups, so we need to restrict the group.
            // We assume the group of the first processor listed for this node.
            // Processors from outside this group will not be assigned for this thread.
            // Normally this won't be an issue because windows used to assign NUMA nodes
            // such that they cannot span processor groups. However, since Windows 10
            // Build 20348 the behaviour changed, so there's a small window of versions
            // between this and Windows 11 that might exhibit problems with not all
            // processors being utilized.
            //
            // We handle this in NumaConfig::from_system by manually splitting the
            // nodes when we detect that there is no function to set affinity spanning
            // processor nodes. This is required because otherwise our thread distribution
            // code may produce suboptimal results.
            //
            // See https://learn.microsoft.com/en-us/windows/win32/procthread/numa-support
            GROUP_AFFINITY affinity;
            std::memset(&affinity, 0, sizeof(GROUP_AFFINITY));
            // We use an ordered set to be sure to get the smallest cpu number here.
            const size_t forcedProcGroupIndex = *(nodes[n].begin()) / WIN_PROCESSOR_GROUP_SIZE;
            affinity.Group                    = static_cast<WORD>(forcedProcGroupIndex);
            for (CpuIndex c : nodes[n])
            {
                const size_t procGroupIndex     = c / WIN_PROCESSOR_GROUP_SIZE;
                const size_t idxWithinProcGroup = c % WIN_PROCESSOR_GROUP_SIZE;
                // We skip processors that are not in the same processor group.
                // If everything was set up correctly this will never be an issue,
                // but we have to account for bad NUMA node specification.
                if (procGroupIndex != forcedProcGroupIndex)
                    continue;

                affinity.Mask |= KAFFINITY(1) << idxWithinProcGroup;
            }

            HANDLE hThread = GetCurrentThread();

            const BOOL status = SetThreadGroupAffinity(hThread, &affinity, nullptr);
            if (status == 0)
                std::exit(EXIT_FAILURE);

            // We yield this thread just to be sure it gets rescheduled. This is
            // defensive, allowed because this code is not performance critical.
            SwitchToThread();
        }
#endif

        return NumaReplicatedAccessToken(n);
    }

    template<typename FuncT>
    void execute_on_numa_node(NumaIndex n, FuncT&& f) const {
        std::thread th([this, &f, n]() {
            bind_current_thread_to_numa_node(n);
            std::forward<FuncT>(f)();
        });

        th.join();
    }

   private:
    std::vector<std::set<CpuIndex>> nodes;
    std::map<CpuIndex, NumaIndex>   nodeByCpu;
    CpuIndex                        highestCpuIndex;

    bool customAffinity;

    static NumaConfig empty() { return NumaConfig(EmptyNodeTag{}); }

    struct EmptyNodeTag {};

    NumaConfig(EmptyNodeTag) :
        highestCpuIndex(0),
        customAffinity(false) {}

    void remove_empty_numa_nodes() {
        std::vector<std::set<CpuIndex>> newNodes;
        for (auto&& cpus : nodes)
            if (!cpus.empty())
                newNodes.emplace_back(std::move(cpus));
        nodes = std::move(newNodes);
    }

    // Returns true if successful
    // Returns false if failed, i.e. when the cpu is already present
    //                          strong guarantee, the structure remains unmodified
    bool add_cpu_to_node(NumaIndex n, CpuIndex c) {
        if (is_cpu_assigned(c))
            return false;

        while (nodes.size() <= n)
            nodes.emplace_back();

        nodes[n].insert(c);
        nodeByCpu[c] = n;

        if (c > highestCpuIndex)
            highestCpuIndex = c;

        return true;
    }

    // Returns true if successful
    // Returns false if failed, i.e. when any of the cpus is already present
    //                          strong guarantee, the structure remains unmodified
    bool add_cpu_range_to_node(NumaIndex n, CpuIndex cfirst, CpuIndex clast) {
        for (CpuIndex c = cfirst; c <= clast; ++c)
            if (is_cpu_assigned(c))
                return false;

        while (nodes.size() <= n)
            nodes.emplace_back();

        for (CpuIndex c = cfirst; c <= clast; ++c)
        {
            nodes[n].insert(c);
            nodeByCpu[c] = n;
        }

        if (clast > highestCpuIndex)
            highestCpuIndex = clast;

        return true;
    }

    static std::vector<size_t> indices_from_shortened_string(const std::string& s) {
        std::vector<size_t> indices;

        if (s.empty())
            return indices;

        for (const std::string& ss : split(s, ","))
        {
            if (ss.empty())
                continue;

            auto parts = split(ss, "-");
            if (parts.size() == 1)
            {
                const CpuIndex c = CpuIndex{str_to_size_t(parts[0])};
                indices.emplace_back(c);
            }
            else if (parts.size() == 2)
            {
                const CpuIndex cfirst = CpuIndex{str_to_size_t(parts[0])};
                const CpuIndex clast  = CpuIndex{str_to_size_t(parts[1])};
                for (size_t c = cfirst; c <= clast; ++c)
                {
                    indices.emplace_back(c);
                }
            }
        }

        return indices;
    }
};

class NumaReplicationContext;

// Instances of this class are tracked by the NumaReplicationContext instance.
// NumaReplicationContext informs all tracked instances when NUMA configuration changes.
class NumaReplicatedBase {
   public:
    NumaReplicatedBase(NumaReplicationContext& ctx);

    NumaReplicatedBase(const NumaReplicatedBase&) = delete;
    NumaReplicatedBase(NumaReplicatedBase&& other) noexcept;

    NumaReplicatedBase& operator=(const NumaReplicatedBase&) = delete;
    NumaReplicatedBase& operator=(NumaReplicatedBase&& other) noexcept;

    virtual void on_numa_config_changed() = 0;
    virtual ~NumaReplicatedBase();

    const NumaConfig& get_numa_config() const;

   private:
    NumaReplicationContext* context;
};

// We force boxing with a unique_ptr. If this becomes an issue due to added
// indirection we may need to add an option for a custom boxing type. When the
// NUMA config changes the value stored at the index 0 is replicated to other nodes.
template<typename T>
class NumaReplicated: public NumaReplicatedBase {
   public:
    using ReplicatorFuncType = std::function<T(const T&)>;

    NumaReplicated(NumaReplicationContext& ctx) :
        NumaReplicatedBase(ctx) {
        replicate_from(T{});
    }

    NumaReplicated(NumaReplicationContext& ctx, T&& source) :
        NumaReplicatedBase(ctx) {
        replicate_from(std::move(source));
    }

    NumaReplicated(const NumaReplicated&) = delete;
    NumaReplicated(NumaReplicated&& other) noexcept :
        NumaReplicatedBase(std::move(other)),
        instances(std::exchange(other.instances, {})) {}

    NumaReplicated& operator=(const NumaReplicated&) = delete;
    NumaReplicated& operator=(NumaReplicated&& other) noexcept {
        NumaReplicatedBase::operator=(*this, std::move(other));
        instances = std::exchange(other.instances, {});

        return *this;
    }

    NumaReplicated& operator=(T&& source) {
        replicate_from(std::move(source));

        return *this;
    }

    ~NumaReplicated() override = default;

    const T& operator[](NumaReplicatedAccessToken token) const {
        assert(token.get_numa_index() < instances.size());
        return *(instances[token.get_numa_index()]);
    }

    const T& operator*() const { return *(instances[0]); }

    const T* operator->() const { return instances[0].get(); }

    template<typename FuncT>
    void modify_and_replicate(FuncT&& f) {
        auto source = std::move(instances[0]);
        std::forward<FuncT>(f)(*source);
        replicate_from(std::move(*source));
    }

    void on_numa_config_changed() override {
        // Use the first one as the source. It doesn't matter which one we use,
        // because they all must be identical, but the first one is guaranteed to exist.
        auto source = std::move(instances[0]);
        replicate_from(std::move(*source));
    }

   private:
    std::vector<std::unique_ptr<T>> instances;

    void replicate_from(T&& source) {
        instances.clear();

        const NumaConfig& cfg = get_numa_config();
        if (cfg.requires_memory_replication())
        {
            for (NumaIndex n = 0; n < cfg.num_numa_nodes(); ++n)
            {
                cfg.execute_on_numa_node(
                  n, [this, &source]() { instances.emplace_back(std::make_unique<T>(source)); });
            }
        }
        else
        {
            assert(cfg.num_numa_nodes() == 1);
            // We take advantage of the fact that replication is not required
            // and reuse the source value, avoiding one copy operation.
            instances.emplace_back(std::make_unique<T>(std::move(source)));
        }
    }
};

// We force boxing with a unique_ptr. If this becomes an issue due to added
// indirection we may need to add an option for a custom boxing type.
template<typename T>
class LazyNumaReplicated: public NumaReplicatedBase {
   public:
    using ReplicatorFuncType = std::function<T(const T&)>;

    LazyNumaReplicated(NumaReplicationContext& ctx) :
        NumaReplicatedBase(ctx) {
        prepare_replicate_from(T{});
    }

    LazyNumaReplicated(NumaReplicationContext& ctx, T&& source) :
        NumaReplicatedBase(ctx) {
        prepare_replicate_from(std::move(source));
    }

    LazyNumaReplicated(const LazyNumaReplicated&) = delete;
    LazyNumaReplicated(LazyNumaReplicated&& other) noexcept :
        NumaReplicatedBase(std::move(other)),
        instances(std::exchange(other.instances, {})) {}

    LazyNumaReplicated& operator=(const LazyNumaReplicated&) = delete;
    LazyNumaReplicated& operator=(LazyNumaReplicated&& other) noexcept {
        NumaReplicatedBase::operator=(*this, std::move(other));
        instances = std::exchange(other.instances, {});

        return *this;
    }

    LazyNumaReplicated& operator=(T&& source) {
        prepare_replicate_from(std::move(source));

        return *this;
    }

    ~LazyNumaReplicated() override = default;

    const T& operator[](NumaReplicatedAccessToken token) const {
        assert(token.get_numa_index() < instances.size());
        ensure_present(token.get_numa_index());
        return *(instances[token.get_numa_index()]);
    }

    const T& operator*() const { return *(instances[0]); }

    const T* operator->() const { return instances[0].get(); }

    template<typename FuncT>
    void modify_and_replicate(FuncT&& f) {
        auto source = std::move(instances[0]);
        std::forward<FuncT>(f)(*source);
        prepare_replicate_from(std::move(*source));
    }

    void on_numa_config_changed() override {
        // Use the first one as the source. It doesn't matter which one we use,
        // because they all must be identical, but the first one is guaranteed to exist.
        auto source = std::move(instances[0]);
        prepare_replicate_from(std::move(*source));
    }

   private:
    mutable std::vector<std::unique_ptr<T>> instances;
    mutable std::mutex                      mutex;

    void ensure_present(NumaIndex idx) const {
        assert(idx < instances.size());

        if (instances[idx] != nullptr)
            return;

        assert(idx != 0);

        std::unique_lock<std::mutex> lock(mutex);
        // Check again for races.
        if (instances[idx] != nullptr)
            return;

        const NumaConfig& cfg = get_numa_config();
        cfg.execute_on_numa_node(
          idx, [this, idx]() { instances[idx] = std::make_unique<T>(*instances[0]); });
    }

    void prepare_replicate_from(T&& source) {
        instances.clear();

        const NumaConfig& cfg = get_numa_config();
        if (cfg.requires_memory_replication())
        {
            assert(cfg.num_numa_nodes() > 0);

            // We just need to make sure the first instance is there.
            // Note that we cannot move here as we need to reallocate the data
            // on the correct NUMA node.
            cfg.execute_on_numa_node(
              0, [this, &source]() { instances.emplace_back(std::make_unique<T>(source)); });

            // Prepare others for lazy init.
            instances.resize(cfg.num_numa_nodes());
        }
        else
        {
            assert(cfg.num_numa_nodes() == 1);
            // We take advantage of the fact that replication is not required
            // and reuse the source value, avoiding one copy operation.
            instances.emplace_back(std::make_unique<T>(std::move(source)));
        }
    }
};

class NumaReplicationContext {
   public:
    NumaReplicationContext(NumaConfig&& cfg) :
        config(std::move(cfg)) {}

    NumaReplicationContext(const NumaReplicationContext&) = delete;
    NumaReplicationContext(NumaReplicationContext&&)      = delete;

    NumaReplicationContext& operator=(const NumaReplicationContext&) = delete;
    NumaReplicationContext& operator=(NumaReplicationContext&&)      = delete;

    ~NumaReplicationContext() {
        // The context must outlive replicated objects
        if (!trackedReplicatedObjects.empty())
            std::exit(EXIT_FAILURE);
    }

    void attach(NumaReplicatedBase* obj) {
        assert(trackedReplicatedObjects.count(obj) == 0);
        trackedReplicatedObjects.insert(obj);
    }

    void detach(NumaReplicatedBase* obj) {
        assert(trackedReplicatedObjects.count(obj) == 1);
        trackedReplicatedObjects.erase(obj);
    }

    // oldObj may be invalid at this point
    void move_attached([[maybe_unused]] NumaReplicatedBase* oldObj, NumaReplicatedBase* newObj) {
        assert(trackedReplicatedObjects.count(oldObj) == 1);
        assert(trackedReplicatedObjects.count(newObj) == 0);
        trackedReplicatedObjects.erase(oldObj);
        trackedReplicatedObjects.insert(newObj);
    }

    void set_numa_config(NumaConfig&& cfg) {
        config = std::move(cfg);
        for (auto&& obj : trackedReplicatedObjects)
            obj->on_numa_config_changed();
    }

    const NumaConfig& get_numa_config() const { return config; }

   private:
    NumaConfig config;

    // std::set uses std::less by default, which is required for pointer comparison
    std::set<NumaReplicatedBase*> trackedReplicatedObjects;
};

inline NumaReplicatedBase::NumaReplicatedBase(NumaReplicationContext& ctx) :
    context(&ctx) {
    context->attach(this);
}

inline NumaReplicatedBase::NumaReplicatedBase(NumaReplicatedBase&& other) noexcept :
    context(std::exchange(other.context, nullptr)) {
    context->move_attached(&other, this);
}

inline NumaReplicatedBase& NumaReplicatedBase::operator=(NumaReplicatedBase&& other) noexcept {
    context = std::exchange(other.context, nullptr);

    context->move_attached(&other, this);

    return *this;
}

inline NumaReplicatedBase::~NumaReplicatedBase() {
    if (context != nullptr)
        context->detach(this);
}

inline const NumaConfig& NumaReplicatedBase::get_numa_config() const {
    return context->get_numa_config();
}

}  // namespace Stockfish

#endif  // #ifndef NUMA_H_INCLUDED
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef SCORE_H_INCLUDED
#define SCORE_H_INCLUDED

namespace Stockfish {

class Position;

class Score {
   public:
    struct Mate {
        int plies;
    };

    struct Tablebase {
        int  plies;
        bool win;
    };

    struct InternalUnits {
        int value;
    };

    Score() = default;
    Score(Value v, const Position& pos);

    template<typename T>
    bool is() const {
        return std::holds_alternative<T>(score);
    }

    template<typename T>
    T get() const {
        return std::get<T>(score);
    }

    template<typename F>
    decltype(auto) visit(F&& f) const {
        return std::visit(std::forward<F>(f), score);
    }

   private:
    std::variant<Mate, Tablebase, InternalUnits> score;
};

}

#endif  // #ifndef SCORE_H_INCLUDED
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef TIMEMAN_H_INCLUDED
#define TIMEMAN_H_INCLUDED

namespace Stockfish {

class OptionsMap;

namespace Search {
struct LimitsType;
}

// The TimeManagement class computes the optimal time to think depending on
// the maximum available time, the game move number, and other parameters.
class TimeManagement {
   public:
    void init(Search::LimitsType& limits,
              Color               us,
              int                 ply,
              const OptionsMap&   options,
              double&             originalTimeAdjust);

    TimePoint optimum() const;
    TimePoint maximum() const;
    template<typename FUNC>
    TimePoint elapsed(FUNC nodes) const {
        return useNodesTime ? TimePoint(nodes()) : elapsed_time();
    }
    TimePoint elapsed_time() const { return now() - startTime; };

    void clear();
    void advance_nodes_time(std::int64_t nodes);

   private:
    TimePoint startTime;
    TimePoint optimumTime;
    TimePoint maximumTime;

    std::int64_t availableNodes = -1;     // When in 'nodes as time' mode
    bool         useNodesTime   = false;  // True if we are in 'nodes as time' mode
};

}  // namespace Stockfish

#endif  // #ifndef TIMEMAN_H_INCLUDED

/**
 * @file incbin.h
 * @author Dale Weiler
 * @brief Utility for including binary files
 *
 * Facilities for including binary files into the current translation unit
 * and making use of them externally in other translation units.
 */
#ifndef INCBIN_HDR
#define INCBIN_HDR
#if   defined(__AVX512BW__) || \
      defined(__AVX512CD__) || \
      defined(__AVX512DQ__) || \
      defined(__AVX512ER__) || \
      defined(__AVX512PF__) || \
      defined(__AVX512VL__) || \
      defined(__AVX512F__)
# define INCBIN_ALIGNMENT_INDEX 6
#elif defined(__AVX__)      || \
      defined(__AVX2__)
# define INCBIN_ALIGNMENT_INDEX 5
#elif defined(__SSE__)      || \
      defined(__SSE2__)     || \
      defined(__SSE3__)     || \
      defined(__SSSE3__)    || \
      defined(__SSE4_1__)   || \
      defined(__SSE4_2__)   || \
      defined(__neon__)
# define INCBIN_ALIGNMENT_INDEX 4
#elif ULONG_MAX != 0xffffffffu
# define INCBIN_ALIGNMENT_INDEX 3
# else
# define INCBIN_ALIGNMENT_INDEX 2
#endif

/* Lookup table of (1 << n) where `n' is `INCBIN_ALIGNMENT_INDEX' */
#define INCBIN_ALIGN_SHIFT_0 1
#define INCBIN_ALIGN_SHIFT_1 2
#define INCBIN_ALIGN_SHIFT_2 4
#define INCBIN_ALIGN_SHIFT_3 8
#define INCBIN_ALIGN_SHIFT_4 16
#define INCBIN_ALIGN_SHIFT_5 32
#define INCBIN_ALIGN_SHIFT_6 64

/* Actual alignment value */
#define INCBIN_ALIGNMENT \
    INCBIN_CONCATENATE( \
        INCBIN_CONCATENATE(INCBIN_ALIGN_SHIFT, _), \
        INCBIN_ALIGNMENT_INDEX)

/* Stringize */
#define INCBIN_STR(X) \
    #X
#define INCBIN_STRINGIZE(X) \
    INCBIN_STR(X)
/* Concatenate */
#define INCBIN_CAT(X, Y) \
    X ## Y
#define INCBIN_CONCATENATE(X, Y) \
    INCBIN_CAT(X, Y)
/* Deferred macro expansion */
#define INCBIN_EVAL(X) \
    X
#define INCBIN_INVOKE(N, ...) \
    INCBIN_EVAL(N(__VA_ARGS__))

/* Green Hills uses a different directive for including binary data */
#if defined(__ghs__)
#  if (__ghs_asm == 2)
#    define INCBIN_MACRO ".file"
/* Or consider the ".myrawdata" entry in the ld file */
#  else
#    define INCBIN_MACRO "\tINCBIN"
#  endif
#else
#  define INCBIN_MACRO ".incbin"
#endif

#ifndef _MSC_VER
#  define INCBIN_ALIGN \
    __attribute__((aligned(INCBIN_ALIGNMENT)))
#else
#  define INCBIN_ALIGN __declspec(align(INCBIN_ALIGNMENT))
#endif

#if defined(__arm__) || /* GNU C and RealView */ \
    defined(__arm) || /* Diab */ \
    defined(_ARM) /* ImageCraft */
#  define INCBIN_ARM
#endif

#ifdef __GNUC__
/* Utilize .balign where supported */
#  define INCBIN_ALIGN_HOST ".balign " INCBIN_STRINGIZE(INCBIN_ALIGNMENT) "\n"
#  define INCBIN_ALIGN_BYTE ".balign 1\n"
#elif defined(INCBIN_ARM)
/*
 * On arm assemblers, the alignment value is calculated as (1 << n) where `n' is
 * the shift count. This is the value passed to `.align'
 */
#  define INCBIN_ALIGN_HOST ".align " INCBIN_STRINGIZE(INCBIN_ALIGNMENT_INDEX) "\n"
#  define INCBIN_ALIGN_BYTE ".align 0\n"
#else
/* We assume other inline assembler's treat `.align' as `.balign' */
#  define INCBIN_ALIGN_HOST ".align " INCBIN_STRINGIZE(INCBIN_ALIGNMENT) "\n"
#  define INCBIN_ALIGN_BYTE ".align 1\n"
#endif

/* INCBIN_CONST is used by incbin.c generated files */
#if defined(__cplusplus)
#  define INCBIN_EXTERNAL extern "C"
#  define INCBIN_CONST    extern const
#else
#  define INCBIN_EXTERNAL extern
#  define INCBIN_CONST    const
#endif

/**
 * @brief Optionally override the linker section into which data is emitted.
 *
 * @warning If you use this facility, you'll have to deal with platform-specific linker output
 * section naming on your own
 *
 * Overriding the default linker output section, e.g for esp8266/Arduino:
 * @code
 * #define INCBIN_OUTPUT_SECTION ".irom.text"
 * INCBIN(Foo, "foo.txt");
 * // Data is emitted into program memory that never gets copied to RAM
 * @endcode
 */
#if !defined(INCBIN_OUTPUT_SECTION)
#  if defined(__APPLE__)
#    define INCBIN_OUTPUT_SECTION         ".const_data"
#  else
#    define INCBIN_OUTPUT_SECTION         ".rodata"
#  endif
#endif

#if defined(__APPLE__)
/* The directives are different for Apple-branded compilers */
#  define INCBIN_SECTION         INCBIN_OUTPUT_SECTION "\n"
#  define INCBIN_GLOBAL(NAME)    ".globl " INCBIN_MANGLE INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME "\n"
#  define INCBIN_INT             ".long "
#  define INCBIN_MANGLE          "_"
#  define INCBIN_BYTE            ".byte "
#  define INCBIN_TYPE(...)
#else
#  define INCBIN_SECTION         ".section " INCBIN_OUTPUT_SECTION "\n"
#  define INCBIN_GLOBAL(NAME)    ".global " INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME "\n"
#  if defined(__ghs__)
#    define INCBIN_INT           ".word "
#  else
#    define INCBIN_INT           ".int "
#  endif
#  if defined(__USER_LABEL_PREFIX__)
#    define INCBIN_MANGLE        INCBIN_STRINGIZE(__USER_LABEL_PREFIX__)
#  else
#    define INCBIN_MANGLE        ""
#  endif
#  if defined(INCBIN_ARM)
/* On arm assemblers, `@' is used as a line comment token */
#    define INCBIN_TYPE(NAME)    ".type " INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME ", %object\n"
#  elif defined(__MINGW32__) || defined(__MINGW64__)
/* Mingw doesn't support this directive either */
#    define INCBIN_TYPE(NAME)
#  else
/* It's safe to use `@' on other architectures */
#    define INCBIN_TYPE(NAME)    ".type " INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME ", @object\n"
#  endif
#  define INCBIN_BYTE            ".byte "
#endif

/* List of style types used for symbol names */
#define INCBIN_STYLE_CAMEL 0
#define INCBIN_STYLE_SNAKE 1

/**
 * @brief Specify the prefix to use for symbol names.
 *
 * By default this is `g', producing symbols of the form:
 * @code
 * INCBIN(Foo, "foo.txt");
 *
 * // Now you have the following symbols:
 * // const unsigned char gFooData[];
 * // const unsigned char *const gFooEnd;
 * // const unsigned int gFooSize;
 * @endcode
 *
 * If however you specify a prefix before including: e.g:
 * @code
 * #define INCBIN_PREFIX incbin
 * INCBIN(Foo, "foo.txt");
 *
 * // Now you have the following symbols instead:
 * // const unsigned char incbinFooData[];
 * // const unsigned char *const incbinFooEnd;
 * // const unsigned int incbinFooSize;
 * @endcode
 */
#if !defined(INCBIN_PREFIX)
#  define INCBIN_PREFIX g
#endif

/**
 * @brief Specify the style used for symbol names.
 *
 * Possible options are
 * - INCBIN_STYLE_CAMEL "CamelCase"
 * - INCBIN_STYLE_SNAKE "snake_case"
 *
 * Default option is *INCBIN_STYLE_CAMEL* producing symbols of the form:
 * @code
 * INCBIN(Foo, "foo.txt");
 *
 * // Now you have the following symbols:
 * // const unsigned char <prefix>FooData[];
 * // const unsigned char *const <prefix>FooEnd;
 * // const unsigned int <prefix>FooSize;
 * @endcode
 *
 * If however you specify a style before including: e.g:
 * @code
 * #define INCBIN_STYLE INCBIN_STYLE_SNAKE
 * INCBIN(foo, "foo.txt");
 *
 * // Now you have the following symbols:
 * // const unsigned char <prefix>foo_data[];
 * // const unsigned char *const <prefix>foo_end;
 * // const unsigned int <prefix>foo_size;
 * @endcode
 */
#if !defined(INCBIN_STYLE)
#  define INCBIN_STYLE INCBIN_STYLE_CAMEL
#endif

/* Style lookup tables */
#define INCBIN_STYLE_0_DATA Data
#define INCBIN_STYLE_0_END End
#define INCBIN_STYLE_0_SIZE Size
#define INCBIN_STYLE_1_DATA _data
#define INCBIN_STYLE_1_END _end
#define INCBIN_STYLE_1_SIZE _size

/* Style lookup: returning identifier */
#define INCBIN_STYLE_IDENT(TYPE) \
    INCBIN_CONCATENATE( \
        INCBIN_STYLE_, \
        INCBIN_CONCATENATE( \
            INCBIN_EVAL(INCBIN_STYLE), \
            INCBIN_CONCATENATE(_, TYPE)))

/* Style lookup: returning string literal */
#define INCBIN_STYLE_STRING(TYPE) \
    INCBIN_STRINGIZE( \
        INCBIN_STYLE_IDENT(TYPE)) \

/* Generate the global labels by indirectly invoking the macro
 * with our style type and concatenate the name against them. */
#define INCBIN_GLOBAL_LABELS(NAME, TYPE) \
    INCBIN_INVOKE( \
        INCBIN_GLOBAL, \
        INCBIN_CONCATENATE( \
            NAME, \
            INCBIN_INVOKE( \
                INCBIN_STYLE_IDENT, \
                TYPE))) \
    INCBIN_INVOKE( \
        INCBIN_TYPE, \
        INCBIN_CONCATENATE( \
            NAME, \
            INCBIN_INVOKE( \
                INCBIN_STYLE_IDENT, \
                TYPE)))

/**
 * @brief Externally reference binary data included in another translation unit.
 *
 * Produces three external symbols that reference the binary data included in
 * another translation unit.
 *
 * The symbol names are a concatenation of `INCBIN_PREFIX' before *NAME*; with
 * "Data", as well as "End" and "Size" after. An example is provided below.
 *
 * @param NAME The name given for the binary data
 *
 * @code
 * INCBIN_EXTERN(Foo);
 *
 * // Now you have the following symbols:
 * // extern const unsigned char <prefix>FooData[];
 * // extern const unsigned char *const <prefix>FooEnd;
 * // extern const unsigned int <prefix>FooSize;
 * @endcode
 */
#define INCBIN_EXTERN(NAME) \
    INCBIN_EXTERNAL const INCBIN_ALIGN unsigned char \
        INCBIN_CONCATENATE( \
            INCBIN_CONCATENATE(INCBIN_PREFIX, NAME), \
            INCBIN_STYLE_IDENT(DATA))[]; \
    INCBIN_EXTERNAL const INCBIN_ALIGN unsigned char *const \
    INCBIN_CONCATENATE( \
        INCBIN_CONCATENATE(INCBIN_PREFIX, NAME), \
        INCBIN_STYLE_IDENT(END)); \
    INCBIN_EXTERNAL const unsigned int \
        INCBIN_CONCATENATE( \
            INCBIN_CONCATENATE(INCBIN_PREFIX, NAME), \
            INCBIN_STYLE_IDENT(SIZE))

/**
 * @brief Include a binary file into the current translation unit.
 *
 * Includes a binary file into the current translation unit, producing three symbols
 * for objects that encode the data and size respectively.
 *
 * The symbol names are a concatenation of `INCBIN_PREFIX' before *NAME*; with
 * "Data", as well as "End" and "Size" after. An example is provided below.
 *
 * @param NAME The name to associate with this binary data (as an identifier.)
 * @param FILENAME The file to include (as a string literal.)
 *
 * @code
 * INCBIN(Icon, "icon.png");
 *
 * // Now you have the following symbols:
 * // const unsigned char <prefix>IconData[];
 * // const unsigned char *const <prefix>IconEnd;
 * // const unsigned int <prefix>IconSize;
 * @endcode
 *
 * @warning This must be used in global scope
 * @warning The identifiers may be different if INCBIN_STYLE is not default
 *
 * To externally reference the data included by this in another translation unit
 * please see INCBIN_EXTERN.
 */
#ifdef _MSC_VER
#define INCBIN(NAME, FILENAME) \
    INCBIN_EXTERN(NAME)
#else
#define INCBIN(NAME, FILENAME) \
    __asm__(INCBIN_SECTION \
            INCBIN_GLOBAL_LABELS(NAME, DATA) \
            INCBIN_ALIGN_HOST \
            INCBIN_MANGLE INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME INCBIN_STYLE_STRING(DATA) ":\n" \
            INCBIN_MACRO " \"" FILENAME "\"\n" \
            INCBIN_GLOBAL_LABELS(NAME, END) \
            INCBIN_ALIGN_BYTE \
            INCBIN_MANGLE INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME INCBIN_STYLE_STRING(END) ":\n" \
                INCBIN_BYTE "1\n" \
            INCBIN_GLOBAL_LABELS(NAME, SIZE) \
            INCBIN_ALIGN_HOST \
            INCBIN_MANGLE INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME INCBIN_STYLE_STRING(SIZE) ":\n" \
                INCBIN_INT INCBIN_MANGLE INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME INCBIN_STYLE_STRING(END) " - " \
                           INCBIN_MANGLE INCBIN_STRINGIZE(INCBIN_PREFIX) #NAME INCBIN_STYLE_STRING(DATA) "\n" \
            INCBIN_ALIGN_HOST \
            ".text\n" \
    ); \
    INCBIN_EXTERN(NAME)

#endif
#endif
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
```

```cpp
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef TUNE_H_INCLUDED
#define TUNE_H_INCLUDED

namespace Stockfish {

class OptionsMap;

using Range    = std::pair<int, int>;  // Option's min-max values
using RangeFun = Range(int);

// Default Range function, to calculate Option's min-max values
inline Range default_range(int v) { return v > 0 ? Range(0, 2 * v) : Range(2 * v, 0); }

struct SetRange {
    explicit SetRange(RangeFun f) :
        fun(f) {}
    SetRange(int min, int max) :
        fun(nullptr),
        range(min, max) {}
    Range operator()(int v) const { return fun ? fun(v) : range; }

    RangeFun* fun;
    Range     range;
};

#define SetDefaultRange SetRange(default_range)

// Tune class implements the 'magic' code that makes the setup of a fishtest tuning
// session as easy as it can be. Mainly you have just to remove const qualifiers
// from the variables you want to tune and flag them for tuning, so if you have:
//
//   const Value myValue[][2] = { { V(100), V(20) }, { V(7), V(78) } };
//
// If you have a my_post_update() function to run after values have been updated,
// and a my_range() function to set custom Option's min-max values, then you just
// remove the 'const' qualifiers and write somewhere below in the file:
//
//   TUNE(SetRange(my_range), myValue, my_post_update);
//
// You can also set the range directly, and restore the default at the end
//
//   TUNE(SetRange(-100, 100), myValue, SetDefaultRange);
//
// In case update function is slow and you have many parameters, you can add:
//
//   UPDATE_ON_LAST();
//
// And the values update, including post update function call, will be done only
// once, after the engine receives the last UCI option, that is the one defined
// and created as the last one, so the GUI should send the options in the same
// order in which have been defined.

class Tune {

    using PostUpdate = void();  // Post-update function

    Tune() { read_results(); }
    Tune(const Tune&)           = delete;
    void operator=(const Tune&) = delete;
    void read_results();

    static Tune& instance() {
        static Tune t;
        return t;
    }  // Singleton

    // Use polymorphism to accommodate Entry of different types in the same vector
    struct EntryBase {
        virtual ~EntryBase()       = default;
        virtual void init_option() = 0;
        virtual void read_option() = 0;
    };

    template<typename T>
    struct Entry: public EntryBase {

        static_assert(!std::is_const_v<T>, "Parameter cannot be const!");

        static_assert(std::is_same_v<T, int> || std::is_same_v<T, PostUpdate>,
                      "Parameter type not supported!");

        Entry(const std::string& n, T& v, const SetRange& r) :
            name(n),
            value(v),
            range(r) {}
        void operator=(const Entry&) = delete;  // Because 'value' is a reference
        void init_option() override;
        void read_option() override;

        std::string name;
        T&          value;
        SetRange    range;
    };

    // Our facility to fill the container, each Entry corresponds to a parameter
    // to tune. We use variadic templates to deal with an unspecified number of
    // entries, each one of a possible different type.
    static std::string next(std::string& names, bool pop = true);

    int add(const SetRange&, std::string&&) { return 0; }

    template<typename T, typename... Args>
    int add(const SetRange& range, std::string&& names, T& value, Args&&... args) {
        list.push_back(std::unique_ptr<EntryBase>(new Entry<T>(next(names), value, range)));
        return add(range, std::move(names), args...);
    }

    // Template specialization for arrays: recursively handle multi-dimensional arrays
    template<typename T, size_t N, typename... Args>
    int add(const SetRange& range, std::string&& names, T (&value)[N], Args&&... args) {
        for (size_t i = 0; i < N; i++)
            add(range, next(names, i == N - 1) + "[" + std::to_string(i) + "]", value[i]);
        return add(range, std::move(names), args...);
    }

    // Template specialization for SetRange
    template<typename... Args>
    int add(const SetRange&, std::string&& names, SetRange& value, Args&&... args) {
        return add(value, (next(names), std::move(names)), args...);
    }

    static void make_option(OptionsMap* options, const std::string& n, int v, const SetRange& r);

    std::vector<std::unique_ptr<EntryBase>> list;

   public:
    template<typename... Args>
    static int add(const std::string& names, Args&&... args) {
        return instance().add(SetDefaultRange, names.substr(1, names.size() - 2),
                              args...);  // Remove trailing parenthesis
    }
    static void init(OptionsMap& o) {
        options = &o;
        for (auto& e : instance().list)
            e->init_option();
        read_options();
    }  // Deferred, due to UCIEngine::Options access
    static void read_options() {
        for (auto& e : instance().list)
            e->read_option();
    }

    static bool        update_on_last;
    static OptionsMap* options;
};

// Some macro magic :-) we define a dummy int variable that the compiler initializes calling Tune::add()
#define STRINGIFY(x) #x
#define UNIQUE2(x, y) x##y
#define UNIQUE(x, y) UNIQUE2(x, y)  // Two indirection levels to expand __LINE__
#define TUNE(...) int UNIQUE(p, __LINE__) = Tune::add(STRINGIFY((__VA_ARGS__)), __VA_ARGS__)

#define UPDATE_ON_LAST() bool UNIQUE(p, __LINE__) = Tune::update_on_last = true

}  // namespace Stockfish

#endif  // #ifndef TUNE_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef EVALUATE_H_INCLUDED
#define EVALUATE_H_INCLUDED

namespace Stockfish {

class Position;

namespace Eval {

// The default net name MUST follow the format nn-[SHA256 first 12 digits].nnue
// for the build process (profile-build and fishtest) to work. Do not change the
// name of the macro or the location where this macro is defined, as it is used
// in the Makefile/Fishtest.
#define EvalFileDefaultNameBig "nn-1111cefa1111.nnue"
#define EvalFileDefaultNameSmall "nn-37f18f62d772.nnue"

namespace NNUE {
struct Networks;
struct AccumulatorCaches;
}

std::string trace(Position& pos, const Eval::NNUE::Networks& networks);

int   simple_eval(const Position& pos, Color c);
bool  use_smallnet(const Position& pos);
Value evaluate(const NNUE::Networks&          networks,
               const Position&                pos,
               Eval::NNUE::AccumulatorCaches& caches,
               int                            optimism);
}  // namespace Eval

}  // namespace Stockfish

#endif  // #ifndef EVALUATE_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Constants used in NNUE evaluation function

#ifndef NNUE_COMMON_H_INCLUDED
#define NNUE_COMMON_H_INCLUDED

#if defined(USE_AVX2)

#elif defined(USE_SSE41)

#elif defined(USE_SSSE3)

#elif defined(USE_SSE2)

#elif defined(USE_NEON)
#endif

namespace Stockfish::Eval::NNUE {

// Version of the evaluation file
constexpr std::uint32_t Version = 0x7AF32F20u;

// Constant used in evaluation value calculation
constexpr int OutputScale     = 16;
constexpr int WeightScaleBits = 6;

// Size of cache line (in bytes)
constexpr std::size_t CacheLineSize = 64;

constexpr const char        Leb128MagicString[]   = "COMPRESSED_LEB128";
constexpr const std::size_t Leb128MagicStringSize = sizeof(Leb128MagicString) - 1;

// SIMD width (in bytes)
#if defined(USE_AVX2)
constexpr std::size_t SimdWidth = 32;

#elif defined(USE_SSE2)
constexpr std::size_t SimdWidth = 16;

#elif defined(USE_NEON)
constexpr std::size_t SimdWidth = 16;
#endif

constexpr std::size_t MaxSimdWidth = 32;

// Type of input feature after conversion
using TransformedFeatureType = std::uint8_t;
using IndexType              = std::uint32_t;

// Round n up to be a multiple of base
template<typename IntType>
constexpr IntType ceil_to_multiple(IntType n, IntType base) {
    return (n + base - 1) / base * base;
}

// Utility to read an integer (signed or unsigned, any size)
// from a stream in little-endian order. We swap the byte order after the read if
// necessary to return a result with the byte ordering of the compiling machine.
template<typename IntType>
inline IntType read_little_endian(std::istream& stream) {
    IntType result;

    if (IsLittleEndian)
        stream.read(reinterpret_cast<char*>(&result), sizeof(IntType));
    else
    {
        std::uint8_t                  u[sizeof(IntType)];
        std::make_unsigned_t<IntType> v = 0;

        stream.read(reinterpret_cast<char*>(u), sizeof(IntType));
        for (std::size_t i = 0; i < sizeof(IntType); ++i)
            v = (v << 8) | u[sizeof(IntType) - i - 1];

        std::memcpy(&result, &v, sizeof(IntType));
    }

    return result;
}

// Utility to write an integer (signed or unsigned, any size)
// to a stream in little-endian order. We swap the byte order before the write if
// necessary to always write in little-endian order, independently of the byte
// ordering of the compiling machine.
template<typename IntType>
inline void write_little_endian(std::ostream& stream, IntType value) {

    if (IsLittleEndian)
        stream.write(reinterpret_cast<const char*>(&value), sizeof(IntType));
    else
    {
        std::uint8_t                  u[sizeof(IntType)];
        std::make_unsigned_t<IntType> v = value;

        std::size_t i = 0;
        // if constexpr to silence the warning about shift by 8
        if constexpr (sizeof(IntType) > 1)
        {
            for (; i + 1 < sizeof(IntType); ++i)
            {
                u[i] = std::uint8_t(v);
                v >>= 8;
            }
        }
        u[i] = std::uint8_t(v);

        stream.write(reinterpret_cast<char*>(u), sizeof(IntType));
    }
}

// Read integers in bulk from a little-endian stream.
// This reads N integers from stream s and puts them in array out.
template<typename IntType>
inline void read_little_endian(std::istream& stream, IntType* out, std::size_t count) {
    if (IsLittleEndian)
        stream.read(reinterpret_cast<char*>(out), sizeof(IntType) * count);
    else
        for (std::size_t i = 0; i < count; ++i)
            out[i] = read_little_endian<IntType>(stream);
}

// Write integers in bulk to a little-endian stream.
// This takes N integers from array values and writes them on stream s.
template<typename IntType>
inline void write_little_endian(std::ostream& stream, const IntType* values, std::size_t count) {
    if (IsLittleEndian)
        stream.write(reinterpret_cast<const char*>(values), sizeof(IntType) * count);
    else
        for (std::size_t i = 0; i < count; ++i)
            write_little_endian<IntType>(stream, values[i]);
}

// Read N signed integers from the stream s, putting them in the array out.
// The stream is assumed to be compressed using the signed LEB128 format.
// See https://en.wikipedia.org/wiki/LEB128 for a description of the compression scheme.
template<typename IntType>
inline void read_leb_128(std::istream& stream, IntType* out, std::size_t count) {

    // Check the presence of our LEB128 magic string
    char leb128MagicString[Leb128MagicStringSize];
    stream.read(leb128MagicString, Leb128MagicStringSize);
    assert(strncmp(Leb128MagicString, leb128MagicString, Leb128MagicStringSize) == 0);

    static_assert(std::is_signed_v<IntType>, "Not implemented for unsigned types");

    const std::uint32_t BUF_SIZE = 4096;
    std::uint8_t        buf[BUF_SIZE];

    auto bytes_left = read_little_endian<std::uint32_t>(stream);

    std::uint32_t buf_pos = BUF_SIZE;
    for (std::size_t i = 0; i < count; ++i)
    {
        IntType result = 0;
        size_t  shift  = 0;
        do
        {
            if (buf_pos == BUF_SIZE)
            {
                stream.read(reinterpret_cast<char*>(buf), std::min(bytes_left, BUF_SIZE));
                buf_pos = 0;
            }

            std::uint8_t byte = buf[buf_pos++];
            --bytes_left;
            result |= (byte & 0x7f) << shift;
            shift += 7;

            if ((byte & 0x80) == 0)
            {
                out[i] = (sizeof(IntType) * 8 <= shift || (byte & 0x40) == 0)
                         ? result
                         : result | ~((1 << shift) - 1);
                break;
            }
        } while (shift < sizeof(IntType) * 8);
    }

    assert(bytes_left == 0);
}

// Write signed integers to a stream with LEB128 compression.
// This takes N integers from array values, compresses them with
// the LEB128 algorithm and writes the result on the stream s.
// See https://en.wikipedia.org/wiki/LEB128 for a description of the compression scheme.
template<typename IntType>
inline void write_leb_128(std::ostream& stream, const IntType* values, std::size_t count) {

    // Write our LEB128 magic string
    stream.write(Leb128MagicString, Leb128MagicStringSize);

    static_assert(std::is_signed_v<IntType>, "Not implemented for unsigned types");

    std::uint32_t byte_count = 0;
    for (std::size_t i = 0; i < count; ++i)
    {
        IntType      value = values[i];
        std::uint8_t byte;
        do
        {
            byte = value & 0x7f;
            value >>= 7;
            ++byte_count;
        } while ((byte & 0x40) == 0 ? value != 0 : value != -1);
    }

    write_little_endian(stream, byte_count);

    const std::uint32_t BUF_SIZE = 4096;
    std::uint8_t        buf[BUF_SIZE];
    std::uint32_t       buf_pos = 0;

    auto flush = [&]() {
        if (buf_pos > 0)
        {
            stream.write(reinterpret_cast<char*>(buf), buf_pos);
            buf_pos = 0;
        }
    };

    auto write = [&](std::uint8_t byte) {
        buf[buf_pos++] = byte;
        if (buf_pos == BUF_SIZE)
            flush();
    };

    for (std::size_t i = 0; i < count; ++i)
    {
        IntType value = values[i];
        while (true)
        {
            std::uint8_t byte = value & 0x7f;
            value >>= 7;
            if ((byte & 0x40) == 0 ? value == 0 : value == -1)
            {
                write(byte);
                break;
            }
            write(byte | 0x80);
        }
    }

    flush();
}

}  // namespace Stockfish::Eval::NNUE

#endif  // #ifndef NNUE_COMMON_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

//Definition of input features HalfKP of NNUE evaluation function

#ifndef NNUE_FEATURES_HALF_KA_V2_HM_H_INCLUDED
#define NNUE_FEATURES_HALF_KA_V2_HM_H_INCLUDED

namespace Stockfish {
struct StateInfo;
class Position;
}

namespace Stockfish::Eval::NNUE::Features {

// Feature HalfKAv2_hm: Combination of the position of own king and the
// position of pieces. Position mirrored such that king is always on e..h files.
class HalfKAv2_hm {

    // Unique number for each piece type on each square
    enum {
        PS_NONE     = 0,
        PS_W_PAWN   = 0,
        PS_B_PAWN   = 1 * SQUARE_NB,
        PS_W_KNIGHT = 2 * SQUARE_NB,
        PS_B_KNIGHT = 3 * SQUARE_NB,
        PS_W_BISHOP = 4 * SQUARE_NB,
        PS_B_BISHOP = 5 * SQUARE_NB,
        PS_W_ROOK   = 6 * SQUARE_NB,
        PS_B_ROOK   = 7 * SQUARE_NB,
        PS_W_QUEEN  = 8 * SQUARE_NB,
        PS_B_QUEEN  = 9 * SQUARE_NB,
        PS_KING     = 10 * SQUARE_NB,
        PS_NB       = 11 * SQUARE_NB
    };

    static constexpr IndexType PieceSquareIndex[COLOR_NB][PIECE_NB] = {
      // Convention: W - us, B - them
      // Viewed from other side, W and B are reversed
      {PS_NONE, PS_W_PAWN, PS_W_KNIGHT, PS_W_BISHOP, PS_W_ROOK, PS_W_QUEEN, PS_KING, PS_NONE,
       PS_NONE, PS_B_PAWN, PS_B_KNIGHT, PS_B_BISHOP, PS_B_ROOK, PS_B_QUEEN, PS_KING, PS_NONE},
      {PS_NONE, PS_B_PAWN, PS_B_KNIGHT, PS_B_BISHOP, PS_B_ROOK, PS_B_QUEEN, PS_KING, PS_NONE,
       PS_NONE, PS_W_PAWN, PS_W_KNIGHT, PS_W_BISHOP, PS_W_ROOK, PS_W_QUEEN, PS_KING, PS_NONE}};

   public:
    // Feature name
    static constexpr const char* Name = "HalfKAv2_hm(Friend)";

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t HashValue = 0x7f234cb8u;

    // Number of feature dimensions
    static constexpr IndexType Dimensions =
      static_cast<IndexType>(SQUARE_NB) * static_cast<IndexType>(PS_NB) / 2;

#define B(v) (v * PS_NB)
    // clang-format off
    static constexpr int KingBuckets[COLOR_NB][SQUARE_NB] = {
      { B(28), B(29), B(30), B(31), B(31), B(30), B(29), B(28),
        B(24), B(25), B(26), B(27), B(27), B(26), B(25), B(24),
        B(20), B(21), B(22), B(23), B(23), B(22), B(21), B(20),
        B(16), B(17), B(18), B(19), B(19), B(18), B(17), B(16),
        B(12), B(13), B(14), B(15), B(15), B(14), B(13), B(12),
        B( 8), B( 9), B(10), B(11), B(11), B(10), B( 9), B( 8),
        B( 4), B( 5), B( 6), B( 7), B( 7), B( 6), B( 5), B( 4),
        B( 0), B( 1), B( 2), B( 3), B( 3), B( 2), B( 1), B( 0) },
      { B( 0), B( 1), B( 2), B( 3), B( 3), B( 2), B( 1), B( 0),
        B( 4), B( 5), B( 6), B( 7), B( 7), B( 6), B( 5), B( 4),
        B( 8), B( 9), B(10), B(11), B(11), B(10), B( 9), B( 8),
        B(12), B(13), B(14), B(15), B(15), B(14), B(13), B(12),
        B(16), B(17), B(18), B(19), B(19), B(18), B(17), B(16),
        B(20), B(21), B(22), B(23), B(23), B(22), B(21), B(20),
        B(24), B(25), B(26), B(27), B(27), B(26), B(25), B(24),
        B(28), B(29), B(30), B(31), B(31), B(30), B(29), B(28) }
    };
    // clang-format on
#undef B
    // clang-format off
    // Orient a square according to perspective (rotates by 180 for black)
    static constexpr int OrientTBL[COLOR_NB][SQUARE_NB] = {
      { SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1,
        SQ_H1, SQ_H1, SQ_H1, SQ_H1, SQ_A1, SQ_A1, SQ_A1, SQ_A1 },
      { SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8,
        SQ_H8, SQ_H8, SQ_H8, SQ_H8, SQ_A8, SQ_A8, SQ_A8, SQ_A8 }
    };
    // clang-format on

    // Maximum number of simultaneously active features.
    static constexpr IndexType MaxActiveDimensions = 32;
    using IndexList                                = ValueList<IndexType, MaxActiveDimensions>;

    // Index of a feature for a given king position and another piece on some square
    template<Color Perspective>
    static IndexType make_index(Square s, Piece pc, Square ksq);

    // Get a list of indices for active features
    template<Color Perspective>
    static void append_active_indices(const Position& pos, IndexList& active);

    // Get a list of indices for recently changed features
    template<Color Perspective>
    static void
    append_changed_indices(Square ksq, const DirtyPiece& dp, IndexList& removed, IndexList& added);

    // Returns the cost of updating one perspective, the most costly one.
    // Assumes no refresh needed.
    static int update_cost(const StateInfo* st);
    static int refresh_cost(const Position& pos);

    // Returns whether the change stored in this StateInfo means
    // that a full accumulator refresh is required.
    static bool requires_refresh(const StateInfo* st, Color perspective);
};

}  // namespace Stockfish::Eval::NNUE::Features

#endif  // #ifndef NNUE_FEATURES_HALF_KA_V2_HM_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Definition of layer AffineTransform of NNUE evaluation function

#ifndef NNUE_LAYERS_AFFINE_TRANSFORM_H_INCLUDED
#define NNUE_LAYERS_AFFINE_TRANSFORM_H_INCLUDED

/*
  This file contains the definition for a fully connected layer (aka affine transform).

    - expected use-case is for when PaddedInputDimensions == 32 and InputDimensions <= 32.
      - that's why AVX512 is hard to implement
    - expected use-case is small layers
    - inputs are processed in chunks of 4, weights are respectively transposed
    - accumulation happens directly to int32s
*/

namespace Stockfish::Eval::NNUE::Layers {

#if defined(USE_SSSE3) || defined(USE_NEON_DOTPROD)
    #define ENABLE_SEQ_OPT
#endif

// Fallback implementation for older/other architectures.
// Requires the input to be padded to at least 16 values.
#ifndef ENABLE_SEQ_OPT

template<IndexType InputDimensions, IndexType PaddedInputDimensions, IndexType OutputDimensions>
static void affine_transform_non_ssse3(std::int32_t*       output,
                                       const std::int8_t*  weights,
                                       const std::int32_t* biases,
                                       const std::uint8_t* input) {
    #if defined(USE_SSE2) || defined(USE_NEON)
        #if defined(USE_SSE2)
    // At least a multiple of 16, with SSE2.
    constexpr IndexType NumChunks   = ceil_to_multiple<IndexType>(InputDimensions, 16) / 16;
    const __m128i       Zeros       = _mm_setzero_si128();
    const auto          inputVector = reinterpret_cast<const __m128i*>(input);

        #elif defined(USE_NEON)
    constexpr IndexType NumChunks   = ceil_to_multiple<IndexType>(InputDimensions, 16) / 16;
    const auto          inputVector = reinterpret_cast<const int8x8_t*>(input);
        #endif

    for (IndexType i = 0; i < OutputDimensions; ++i)
    {
        const IndexType offset = i * PaddedInputDimensions;

        #if defined(USE_SSE2)
        __m128i    sumLo = _mm_cvtsi32_si128(biases[i]);
        __m128i    sumHi = Zeros;
        const auto row   = reinterpret_cast<const __m128i*>(&weights[offset]);
        for (IndexType j = 0; j < NumChunks; ++j)
        {
            __m128i row_j           = _mm_load_si128(&row[j]);
            __m128i input_j         = _mm_load_si128(&inputVector[j]);
            __m128i extendedRowLo   = _mm_srai_epi16(_mm_unpacklo_epi8(row_j, row_j), 8);
            __m128i extendedRowHi   = _mm_srai_epi16(_mm_unpackhi_epi8(row_j, row_j), 8);
            __m128i extendedInputLo = _mm_unpacklo_epi8(input_j, Zeros);
            __m128i extendedInputHi = _mm_unpackhi_epi8(input_j, Zeros);
            __m128i productLo       = _mm_madd_epi16(extendedRowLo, extendedInputLo);
            __m128i productHi       = _mm_madd_epi16(extendedRowHi, extendedInputHi);
            sumLo                   = _mm_add_epi32(sumLo, productLo);
            sumHi                   = _mm_add_epi32(sumHi, productHi);
        }
        __m128i sum           = _mm_add_epi32(sumLo, sumHi);
        __m128i sumHigh_64    = _mm_shuffle_epi32(sum, _MM_SHUFFLE(1, 0, 3, 2));
        sum                   = _mm_add_epi32(sum, sumHigh_64);
        __m128i sum_second_32 = _mm_shufflelo_epi16(sum, _MM_SHUFFLE(1, 0, 3, 2));
        sum                   = _mm_add_epi32(sum, sum_second_32);
        output[i]             = _mm_cvtsi128_si32(sum);

        #elif defined(USE_NEON)

        int32x4_t  sum = {biases[i]};
        const auto row = reinterpret_cast<const int8x8_t*>(&weights[offset]);
        for (IndexType j = 0; j < NumChunks; ++j)
        {
            int16x8_t product = vmull_s8(inputVector[j * 2], row[j * 2]);
            product           = vmlal_s8(product, inputVector[j * 2 + 1], row[j * 2 + 1]);
            sum               = vpadalq_s16(sum, product);
        }
        output[i] = sum[0] + sum[1] + sum[2] + sum[3];

        #endif
    }
    #else
    std::memcpy(output, biases, sizeof(std::int32_t) * OutputDimensions);

    // Traverse weights in transpose order to take advantage of input sparsity
    for (IndexType i = 0; i < InputDimensions; ++i)
        if (input[i])
        {
            const std::int8_t* w  = &weights[i];
            const int          in = input[i];
            for (IndexType j = 0; j < OutputDimensions; ++j)
                output[j] += w[j * PaddedInputDimensions] * in;
        }
    #endif
}

#endif  // !ENABLE_SEQ_OPT

template<IndexType InDims, IndexType OutDims>
class AffineTransform {
   public:
    // Input/output type
    using InputType  = std::uint8_t;
    using OutputType = std::int32_t;

    // Number of input/output dimensions
    static constexpr IndexType InputDimensions  = InDims;
    static constexpr IndexType OutputDimensions = OutDims;

    static constexpr IndexType PaddedInputDimensions =
      ceil_to_multiple<IndexType>(InputDimensions, MaxSimdWidth);
    static constexpr IndexType PaddedOutputDimensions =
      ceil_to_multiple<IndexType>(OutputDimensions, MaxSimdWidth);

    using OutputBuffer = OutputType[PaddedOutputDimensions];

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t get_hash_value(std::uint32_t prevHash) {
        std::uint32_t hashValue = 0xCC03DAE4u;
        hashValue += OutputDimensions;
        hashValue ^= prevHash >> 1;
        hashValue ^= prevHash << 31;
        return hashValue;
    }

    static constexpr IndexType get_weight_index_scrambled(IndexType i) {
        return (i / 4) % (PaddedInputDimensions / 4) * OutputDimensions * 4
             + i / PaddedInputDimensions * 4 + i % 4;
    }

    static constexpr IndexType get_weight_index(IndexType i) {
#ifdef ENABLE_SEQ_OPT
        return get_weight_index_scrambled(i);
#else
        return i;
#endif
    }

    // Read network parameters
    bool read_parameters(std::istream& stream) {
        read_little_endian<BiasType>(stream, biases, OutputDimensions);
        for (IndexType i = 0; i < OutputDimensions * PaddedInputDimensions; ++i)
            weights[get_weight_index(i)] = read_little_endian<WeightType>(stream);

        return !stream.fail();
    }

    // Write network parameters
    bool write_parameters(std::ostream& stream) const {
        write_little_endian<BiasType>(stream, biases, OutputDimensions);

        for (IndexType i = 0; i < OutputDimensions * PaddedInputDimensions; ++i)
            write_little_endian<WeightType>(stream, weights[get_weight_index(i)]);

        return !stream.fail();
    }
    // Forward propagation
    void propagate(const InputType* input, OutputType* output) const {

#ifdef ENABLE_SEQ_OPT

        if constexpr (OutputDimensions > 1)
        {
    #if defined(USE_AVX512)
            using vec_t = __m512i;
        #define vec_set_32 _mm512_set1_epi32
        #define vec_add_dpbusd_32 Simd::m512_add_dpbusd_epi32
    #elif defined(USE_AVX2)
            using vec_t = __m256i;
        #define vec_set_32 _mm256_set1_epi32
        #define vec_add_dpbusd_32 Simd::m256_add_dpbusd_epi32
    #elif defined(USE_SSSE3)
            using vec_t = __m128i;
        #define vec_set_32 _mm_set1_epi32
        #define vec_add_dpbusd_32 Simd::m128_add_dpbusd_epi32
    #elif defined(USE_NEON_DOTPROD)
            using vec_t = int32x4_t;
        #define vec_set_32 vdupq_n_s32
        #define vec_add_dpbusd_32(acc, a, b) \
            Simd::dotprod_m128_add_dpbusd_epi32(acc, vreinterpretq_s8_s32(a), \
                                                vreinterpretq_s8_s32(b))
    #endif

            static constexpr IndexType OutputSimdWidth = sizeof(vec_t) / sizeof(OutputType);

            static_assert(OutputDimensions % OutputSimdWidth == 0);

            constexpr IndexType NumChunks = ceil_to_multiple<IndexType>(InputDimensions, 8) / 4;
            constexpr IndexType NumRegs   = OutputDimensions / OutputSimdWidth;

            const auto   input32 = reinterpret_cast<const std::int32_t*>(input);
            const vec_t* biasvec = reinterpret_cast<const vec_t*>(biases);
            vec_t        acc[NumRegs];
            for (IndexType k = 0; k < NumRegs; ++k)
                acc[k] = biasvec[k];

            for (IndexType i = 0; i < NumChunks; ++i)
            {
                const vec_t in0 = vec_set_32(input32[i]);
                const auto  col0 =
                  reinterpret_cast<const vec_t*>(&weights[i * OutputDimensions * 4]);

                for (IndexType k = 0; k < NumRegs; ++k)
                    vec_add_dpbusd_32(acc[k], in0, col0[k]);
            }

            vec_t* outptr = reinterpret_cast<vec_t*>(output);
            for (IndexType k = 0; k < NumRegs; ++k)
                outptr[k] = acc[k];

    #undef vec_set_32
    #undef vec_add_dpbusd_32
        }
        else if constexpr (OutputDimensions == 1)
        {
    // We cannot use AVX512 for the last layer because there are only 32 inputs
    // and the buffer is not padded to 64 elements.
    #if defined(USE_AVX2)
            using vec_t = __m256i;
        #define vec_setzero() _mm256_setzero_si256()
        #define vec_set_32 _mm256_set1_epi32
        #define vec_add_dpbusd_32 Simd::m256_add_dpbusd_epi32
        #define vec_hadd Simd::m256_hadd
    #elif defined(USE_SSSE3)
            using vec_t = __m128i;
        #define vec_setzero() _mm_setzero_si128()
        #define vec_set_32 _mm_set1_epi32
        #define vec_add_dpbusd_32 Simd::m128_add_dpbusd_epi32
        #define vec_hadd Simd::m128_hadd
    #elif defined(USE_NEON_DOTPROD)
            using vec_t = int32x4_t;
        #define vec_setzero() vdupq_n_s32(0)
        #define vec_set_32 vdupq_n_s32
        #define vec_add_dpbusd_32(acc, a, b) \
            Simd::dotprod_m128_add_dpbusd_epi32(acc, vreinterpretq_s8_s32(a), \
                                                vreinterpretq_s8_s32(b))
        #define vec_hadd Simd::neon_m128_hadd
    #endif

            const auto inputVector = reinterpret_cast<const vec_t*>(input);

            static constexpr IndexType InputSimdWidth = sizeof(vec_t) / sizeof(InputType);

            static_assert(PaddedInputDimensions % InputSimdWidth == 0);

            constexpr IndexType NumChunks = PaddedInputDimensions / InputSimdWidth;
            vec_t               sum0      = vec_setzero();
            const auto          row0      = reinterpret_cast<const vec_t*>(&weights[0]);

            for (int j = 0; j < int(NumChunks); ++j)
            {
                const vec_t in = inputVector[j];
                vec_add_dpbusd_32(sum0, in, row0[j]);
            }
            output[0] = vec_hadd(sum0, biases[0]);

    #undef vec_setzero
    #undef vec_set_32
    #undef vec_add_dpbusd_32
    #undef vec_hadd
        }
#else
        // Use old implementation for the other architectures.
        affine_transform_non_ssse3<InputDimensions, PaddedInputDimensions, OutputDimensions>(
          output, weights, biases, input);
#endif
    }

   private:
    using BiasType   = OutputType;
    using WeightType = std::int8_t;

    alignas(CacheLineSize) BiasType biases[OutputDimensions];
    alignas(CacheLineSize) WeightType weights[OutputDimensions * PaddedInputDimensions];
};

}  // namespace Stockfish::Eval::NNUE::Layers

#endif  // #ifndef NNUE_LAYERS_AFFINE_TRANSFORM_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Definition of layer AffineTransformSparseInput of NNUE evaluation function
```

```
#ifndef NNUE_LAYERS_AFFINE_TRANSFORM_SPARSE_INPUT_H_INCLUDED
#define NNUE_LAYERS_AFFINE_TRANSFORM_SPARSE_INPUT_H_INCLUDED

/*
  This file contains the definition for a fully connected layer (aka affine transform) with block sparse input.
*/

namespace Stockfish::Eval::NNUE::Layers {

#if (USE_SSSE3 | (USE_NEON >= 8))
alignas(CacheLineSize) static inline const
  std::array<std::array<std::uint16_t, 8>, 256> lookup_indices = []() {
      std::array<std::array<std::uint16_t, 8>, 256> v{};
      for (unsigned i = 0; i < 256; ++i)
      {
          std::uint64_t j = i, k = 0;
          while (j)
              v[i][k++] = pop_lsb(j);
      }
      return v;
  }();

// Find indices of nonzero numbers in an int32_t array
template<const IndexType InputDimensions>
void find_nnz(const std::int32_t* input, std::uint16_t* out, IndexType& count_out) {
    #if defined(USE_SSSE3)
        #if defined(USE_AVX512)
    using vec_t = __m512i;
            #define vec_nnz(a) _mm512_cmpgt_epi32_mask(a, _mm512_setzero_si512())
        #elif defined(USE_AVX2)
    using vec_t = __m256i;
            #if defined(USE_VNNI) && !defined(USE_AVXVNNI)
                #define vec_nnz(a) _mm256_cmpgt_epi32_mask(a, _mm256_setzero_si256())
            #else
                #define vec_nnz(a) \
                    _mm256_movemask_ps( \
                      _mm256_castsi256_ps(_mm256_cmpgt_epi32(a, _mm256_setzero_si256())))
            #endif
        #elif defined(USE_SSSE3)
    using vec_t = __m128i;
            #define vec_nnz(a) \
                _mm_movemask_ps(_mm_castsi128_ps(_mm_cmpgt_epi32(a, _mm_setzero_si128())))
        #endif
    using vec128_t = __m128i;
        #define vec128_zero _mm_setzero_si128()
        #define vec128_set_16(a) _mm_set1_epi16(a)
        #define vec128_load(a) _mm_load_si128(a)
        #define vec128_storeu(a, b) _mm_storeu_si128(a, b)
        #define vec128_add(a, b) _mm_add_epi16(a, b)
    #elif defined(USE_NEON)
    using vec_t                        = uint32x4_t;
    static const std::uint32_t Mask[4] = {1, 2, 4, 8};
        #define vec_nnz(a) vaddvq_u32(vandq_u32(vtstq_u32(a, a), vld1q_u32(Mask)))
    using vec128_t                     = uint16x8_t;
        #define vec128_zero vdupq_n_u16(0)
        #define vec128_set_16(a) vdupq_n_u16(a)
        #define vec128_load(a) vld1q_u16(reinterpret_cast<const std::uint16_t*>(a))
        #define vec128_storeu(a, b) vst1q_u16(reinterpret_cast<std::uint16_t*>(a), b)
        #define vec128_add(a, b) vaddq_u16(a, b)
    #endif
    constexpr IndexType InputSimdWidth = sizeof(vec_t) / sizeof(std::int32_t);
    // Inputs are processed InputSimdWidth at a time and outputs are processed 8 at a time so we process in chunks of max(InputSimdWidth, 8)
    constexpr IndexType ChunkSize       = std::max<IndexType>(InputSimdWidth, 8);
    constexpr IndexType NumChunks       = InputDimensions / ChunkSize;
    constexpr IndexType InputsPerChunk  = ChunkSize / InputSimdWidth;
    constexpr IndexType OutputsPerChunk = ChunkSize / 8;

    const auto     inputVector = reinterpret_cast<const vec_t*>(input);
    IndexType      count       = 0;
    vec128_t       base        = vec128_zero;
    const vec128_t increment   = vec128_set_16(8);
    for (IndexType i = 0; i < NumChunks; ++i)
    {
        // bitmask of nonzero values in this chunk
        unsigned nnz = 0;
        for (IndexType j = 0; j < InputsPerChunk; ++j)
        {
            const vec_t inputChunk = inputVector[i * InputsPerChunk + j];
            nnz |= unsigned(vec_nnz(inputChunk)) << (j * InputSimdWidth);
        }
        for (IndexType j = 0; j < OutputsPerChunk; ++j)
        {
            const auto lookup = (nnz >> (j * 8)) & 0xFF;
            const auto offsets =
              vec128_load(reinterpret_cast<const vec128_t*>(&lookup_indices[lookup]));
            vec128_storeu(reinterpret_cast<vec128_t*>(out + count), vec128_add(base, offsets));
            count += popcount(lookup);
            base = vec128_add(base, increment);
        }
    }
    count_out = count;
}
    #undef vec_nnz
    #undef vec128_zero
    #undef vec128_set_16
    #undef vec128_load
    #undef vec128_storeu
    #undef vec128_add
#endif

// Sparse input implementation
template<IndexType InDims, IndexType OutDims>
class AffineTransformSparseInput {
   public:
    // Input/output type
    using InputType  = std::uint8_t;
    using OutputType = std::int32_t;

    // Number of input/output dimensions
    static constexpr IndexType InputDimensions  = InDims;
    static constexpr IndexType OutputDimensions = OutDims;

    static_assert(OutputDimensions % 16 == 0,
                  "Only implemented for OutputDimensions divisible by 16.");

    static constexpr IndexType PaddedInputDimensions =
      ceil_to_multiple<IndexType>(InputDimensions, MaxSimdWidth);
    static constexpr IndexType PaddedOutputDimensions =
      ceil_to_multiple<IndexType>(OutputDimensions, MaxSimdWidth);

#if (USE_SSSE3 | (USE_NEON >= 8))
    static constexpr IndexType ChunkSize = 4;
#else
    static constexpr IndexType ChunkSize = 1;
#endif

    using OutputBuffer = OutputType[PaddedOutputDimensions];

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t get_hash_value(std::uint32_t prevHash) {
        std::uint32_t hashValue = 0xCC03DAE4u;
        hashValue += OutputDimensions;
        hashValue ^= prevHash >> 1;
        hashValue ^= prevHash << 31;
        return hashValue;
    }

    static constexpr IndexType get_weight_index_scrambled(IndexType i) {
        return (i / ChunkSize) % (PaddedInputDimensions / ChunkSize) * OutputDimensions * ChunkSize
             + i / PaddedInputDimensions * ChunkSize + i % ChunkSize;
    }

    static constexpr IndexType get_weight_index(IndexType i) {
#if (USE_SSSE3 | (USE_NEON >= 8))
        return get_weight_index_scrambled(i);
#else
        return i;
#endif
    }

    // Read network parameters
    bool read_parameters(std::istream& stream) {
        read_little_endian<BiasType>(stream, biases, OutputDimensions);
        for (IndexType i = 0; i < OutputDimensions * PaddedInputDimensions; ++i)
            weights[get_weight_index(i)] = read_little_endian<WeightType>(stream);

        return !stream.fail();
    }

    // Write network parameters
    bool write_parameters(std::ostream& stream) const {
        write_little_endian<BiasType>(stream, biases, OutputDimensions);

        for (IndexType i = 0; i < OutputDimensions * PaddedInputDimensions; ++i)
            write_little_endian<WeightType>(stream, weights[get_weight_index(i)]);

        return !stream.fail();
    }
    // Forward propagation
    void propagate(const InputType* input, OutputType* output) const {

#if (USE_SSSE3 | (USE_NEON >= 8))
    #if defined(USE_AVX512)
        using invec_t  = __m512i;
        using outvec_t = __m512i;
        #define vec_set_32 _mm512_set1_epi32
        #define vec_add_dpbusd_32 Simd::m512_add_dpbusd_epi32
    #elif defined(USE_AVX2)
        using invec_t  = __m256i;
        using outvec_t = __m256i;
        #define vec_set_32 _mm256_set1_epi32
        #define vec_add_dpbusd_32 Simd::m256_add_dpbusd_epi32
    #elif defined(USE_SSSE3)
        using invec_t  = __m128i;
        using outvec_t = __m128i;
        #define vec_set_32 _mm_set1_epi32
        #define vec_add_dpbusd_32 Simd::m128_add_dpbusd_epi32
    #elif defined(USE_NEON_DOTPROD)
        using invec_t  = int8x16_t;
        using outvec_t = int32x4_t;
        #define vec_set_32(a) vreinterpretq_s8_u32(vdupq_n_u32(a))
        #define vec_add_dpbusd_32 Simd::dotprod_m128_add_dpbusd_epi32
    #elif defined(USE_NEON)
        using invec_t  = int8x16_t;
        using outvec_t = int32x4_t;
        #define vec_set_32(a) vreinterpretq_s8_u32(vdupq_n_u32(a))
        #define vec_add_dpbusd_32 Simd::neon_m128_add_dpbusd_epi32
    #endif
        static constexpr IndexType OutputSimdWidth = sizeof(outvec_t) / sizeof(OutputType);

        constexpr IndexType NumChunks = ceil_to_multiple<IndexType>(InputDimensions, 8) / ChunkSize;
        constexpr IndexType NumRegs   = OutputDimensions / OutputSimdWidth;
        std::uint16_t       nnz[NumChunks];
        IndexType           count;

        const auto input32 = reinterpret_cast<const std::int32_t*>(input);

        // Find indices of nonzero 32-bit blocks
        find_nnz<NumChunks>(input32, nnz, count);

        const outvec_t* biasvec = reinterpret_cast<const outvec_t*>(biases);
        outvec_t        acc[NumRegs];
        for (IndexType k = 0; k < NumRegs; ++k)
            acc[k] = biasvec[k];

        for (IndexType j = 0; j < count; ++j)
        {
            const auto    i  = nnz[j];
            const invec_t in = vec_set_32(input32[i]);
            const auto    col =
              reinterpret_cast<const invec_t*>(&weights[i * OutputDimensions * ChunkSize]);
            for (IndexType k = 0; k < NumRegs; ++k)
                vec_add_dpbusd_32(acc[k], in, col[k]);
        }

        outvec_t* outptr = reinterpret_cast<outvec_t*>(output);
        for (IndexType k = 0; k < NumRegs; ++k)
            outptr[k] = acc[k];
    #undef vec_set_32
    #undef vec_add_dpbusd_32
#else
        // Use dense implementation for the other architectures.
        affine_transform_non_ssse3<InputDimensions, PaddedInputDimensions, OutputDimensions>(
          output, weights, biases, input);
#endif
    }

   private:
    using BiasType   = OutputType;
    using WeightType = std::int8_t;

    alignas(CacheLineSize) BiasType biases[OutputDimensions];
    alignas(CacheLineSize) WeightType weights[OutputDimensions * PaddedInputDimensions];
};

}  // namespace Stockfish::Eval::NNUE::Layers

#endif  // #ifndef NNUE_LAYERS_AFFINE_TRANSFORM_SPARSE_INPUT_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Definition of layer ClippedReLU of NNUE evaluation function

#ifndef NNUE_LAYERS_SQR_CLIPPED_RELU_H_INCLUDED
#define NNUE_LAYERS_SQR_CLIPPED_RELU_H_INCLUDED

namespace Stockfish::Eval::NNUE::Layers {

// Clipped ReLU
template<IndexType InDims>
class SqrClippedReLU {
   public:
    // Input/output type
    using InputType  = std::int32_t;
    using OutputType = std::uint8_t;

    // Number of input/output dimensions
    static constexpr IndexType InputDimensions  = InDims;
    static constexpr IndexType OutputDimensions = InputDimensions;
    static constexpr IndexType PaddedOutputDimensions =
      ceil_to_multiple<IndexType>(OutputDimensions, 32);

    using OutputBuffer = OutputType[PaddedOutputDimensions];

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t get_hash_value(std::uint32_t prevHash) {
        std::uint32_t hashValue = 0x538D24C7u;
        hashValue += prevHash;
        return hashValue;
    }

    // Read network parameters
    bool read_parameters(std::istream&) { return true; }

    // Write network parameters
    bool write_parameters(std::ostream&) const { return true; }

    // Forward propagation
    void propagate(const InputType* input, OutputType* output) const {

#if defined(USE_SSE2)
        constexpr IndexType NumChunks = InputDimensions / 16;

        static_assert(WeightScaleBits == 6);
        const auto in  = reinterpret_cast<const __m128i*>(input);
        const auto out = reinterpret_cast<__m128i*>(output);
        for (IndexType i = 0; i < NumChunks; ++i)
        {
            __m128i words0 =
              _mm_packs_epi32(_mm_load_si128(&in[i * 4 + 0]), _mm_load_si128(&in[i * 4 + 1]));
            __m128i words1 =
              _mm_packs_epi32(_mm_load_si128(&in[i * 4 + 2]), _mm_load_si128(&in[i * 4 + 3]));

            // We shift by WeightScaleBits * 2 = 12 and divide by 128
            // which is an additional shift-right of 7, meaning 19 in total.
            // MulHi strips the lower 16 bits so we need to shift out 3 more to match.
            words0 = _mm_srli_epi16(_mm_mulhi_epi16(words0, words0), 3);
            words1 = _mm_srli_epi16(_mm_mulhi_epi16(words1, words1), 3);

            _mm_store_si128(&out[i], _mm_packs_epi16(words0, words1));
        }
        constexpr IndexType Start = NumChunks * 16;

#else
        constexpr IndexType Start = 0;
#endif

        for (IndexType i = Start; i < InputDimensions; ++i)
        {
            output[i] = static_cast<OutputType>(
              // Really should be /127 but we need to make it fast so we right-shift
              // by an extra 7 bits instead. Needs to be accounted for in the trainer.
              std::min(127ll, ((long long) (input[i]) * input[i]) >> (2 * WeightScaleBits + 7)));
        }
    }
};

}  // namespace Stockfish::Eval::NNUE::Layers

#endif  // NNUE_LAYERS_SQR_CLIPPED_RELU_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Definition of layer ClippedReLU of NNUE evaluation function

#ifndef NNUE_LAYERS_CLIPPED_RELU_H_INCLUDED
#define NNUE_LAYERS_CLIPPED_RELU_H_INCLUDED

namespace Stockfish::Eval::NNUE::Layers {

// Clipped ReLU
template<IndexType InDims>
class ClippedReLU {
   public:
    // Input/output type
    using InputType  = std::int32_t;
    using OutputType = std::uint8_t;

    // Number of input/output dimensions
    static constexpr IndexType InputDimensions  = InDims;
    static constexpr IndexType OutputDimensions = InputDimensions;
    static constexpr IndexType PaddedOutputDimensions =
      ceil_to_multiple<IndexType>(OutputDimensions, 32);

    using OutputBuffer = OutputType[PaddedOutputDimensions];

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t get_hash_value(std::uint32_t prevHash) {
        std::uint32_t hashValue = 0x538D24C7u;
        hashValue += prevHash;
        return hashValue;
    }

    // Read network parameters
    bool read_parameters(std::istream&) { return true; }

    // Write network parameters
    bool write_parameters(std::ostream&) const { return true; }

    // Forward propagation
    void propagate(const InputType* input, OutputType* output) const {

#if defined(USE_AVX2)
        if constexpr (InputDimensions % SimdWidth == 0)
        {
            constexpr IndexType NumChunks = InputDimensions / SimdWidth;
            const __m256i       Offsets   = _mm256_set_epi32(7, 3, 6, 2, 5, 1, 4, 0);
            const auto          in        = reinterpret_cast<const __m256i*>(input);
            const auto          out       = reinterpret_cast<__m256i*>(output);
            for (IndexType i = 0; i < NumChunks; ++i)
            {
                const __m256i words0 =
                  _mm256_srli_epi16(_mm256_packus_epi32(_mm256_load_si256(&in[i * 4 + 0]),
                                                        _mm256_load_si256(&in[i * 4 + 1])),
                                    WeightScaleBits);
                const __m256i words1 =
                  _mm256_srli_epi16(_mm256_packus_epi32(_mm256_load_si256(&in[i * 4 + 2]),
                                                        _mm256_load_si256(&in[i * 4 + 3])),
                                    WeightScaleBits);
                _mm256_store_si256(&out[i], _mm256_permutevar8x32_epi32(
                                              _mm256_packs_epi16(words0, words1), Offsets));
            }
        }
        else
        {
            constexpr IndexType NumChunks = InputDimensions / (SimdWidth / 2);
            const auto          in        = reinterpret_cast<const __m128i*>(input);
            const auto          out       = reinterpret_cast<__m128i*>(output);
            for (IndexType i = 0; i < NumChunks; ++i)
            {
                const __m128i words0 = _mm_srli_epi16(
                  _mm_packus_epi32(_mm_load_si128(&in[i * 4 + 0]), _mm_load_si128(&in[i * 4 + 1])),
                  WeightScaleBits);
                const __m128i words1 = _mm_srli_epi16(
                  _mm_packus_epi32(_mm_load_si128(&in[i * 4 + 2]), _mm_load_si128(&in[i * 4 + 3])),
                  WeightScaleBits);
                _mm_store_si128(&out[i], _mm_packs_epi16(words0, words1));
            }
        }
        constexpr IndexType Start = InputDimensions % SimdWidth == 0
                                    ? InputDimensions / SimdWidth * SimdWidth
                                    : InputDimensions / (SimdWidth / 2) * (SimdWidth / 2);

#elif defined(USE_SSE2)
        constexpr IndexType NumChunks = InputDimensions / SimdWidth;

    #ifndef USE_SSE41
        const __m128i k0x80s = _mm_set1_epi8(-128);
    #endif

        const auto in  = reinterpret_cast<const __m128i*>(input);
        const auto out = reinterpret_cast<__m128i*>(output);
        for (IndexType i = 0; i < NumChunks; ++i)
        {
    #if defined(USE_SSE41)
            const __m128i words0 = _mm_srli_epi16(
              _mm_packus_epi32(_mm_load_si128(&in[i * 4 + 0]), _mm_load_si128(&in[i * 4 + 1])),
              WeightScaleBits);
            const __m128i words1 = _mm_srli_epi16(
              _mm_packus_epi32(_mm_load_si128(&in[i * 4 + 2]), _mm_load_si128(&in[i * 4 + 3])),
              WeightScaleBits);
            _mm_store_si128(&out[i], _mm_packs_epi16(words0, words1));
    #else
            const __m128i words0 = _mm_srai_epi16(
              _mm_packs_epi32(_mm_load_si128(&in[i * 4 + 0]), _mm_load_si128(&in[i * 4 + 1])),
              WeightScaleBits);
            const __m128i words1 = _mm_srai_epi16(
              _mm_packs_epi32(_mm_load_si128(&in[i * 4 + 2]), _mm_load_si128(&in[i * 4 + 3])),
              WeightScaleBits);
            const __m128i packedbytes = _mm_packs_epi16(words0, words1);
            _mm_store_si128(&out[i], _mm_subs_epi8(_mm_adds_epi8(packedbytes, k0x80s), k0x80s));
    #endif
        }
        constexpr IndexType Start = NumChunks * SimdWidth;

#elif defined(USE_NEON)
        constexpr IndexType NumChunks = InputDimensions / (SimdWidth / 2);
        const int8x8_t      Zero      = {0};
        const auto          in        = reinterpret_cast<const int32x4_t*>(input);
        const auto          out       = reinterpret_cast<int8x8_t*>(output);
        for (IndexType i = 0; i < NumChunks; ++i)
        {
            int16x8_t  shifted;
            const auto pack = reinterpret_cast<int16x4_t*>(&shifted);
            pack[0]         = vqshrn_n_s32(in[i * 2 + 0], WeightScaleBits);
            pack[1]         = vqshrn_n_s32(in[i * 2 + 1], WeightScaleBits);
            out[i]          = vmax_s8(vqmovn_s16(shifted), Zero);
        }
        constexpr IndexType Start = NumChunks * (SimdWidth / 2);
#else
        constexpr IndexType Start = 0;
#endif

        for (IndexType i = Start; i < InputDimensions; ++i)
        {
            output[i] = static_cast<OutputType>(std::clamp(input[i] >> WeightScaleBits, 0, 127));
        }
    }
};

}  // namespace Stockfish::Eval::NNUE::Layers

#endif  // NNUE_LAYERS_CLIPPED_RELU_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Input features and network structure used in NNUE evaluation function

#ifndef NNUE_ARCHITECTURE_H_INCLUDED
#define NNUE_ARCHITECTURE_H_INCLUDED

namespace Stockfish::Eval::NNUE {

// Input features used in evaluation function
using FeatureSet = Features::HalfKAv2_hm;

// Number of input feature dimensions after conversion
constexpr IndexType TransformedFeatureDimensionsBig = 3072;
constexpr int       L2Big                           = 15;
constexpr int       L3Big                           = 32;

constexpr IndexType TransformedFeatureDimensionsSmall = 128;
constexpr int       L2Small                           = 15;
constexpr int       L3Small                           = 32;

constexpr IndexType PSQTBuckets = 8;
constexpr IndexType LayerStacks = 8;

template<IndexType L1, int L2, int L3>
struct NetworkArchitecture {
    static constexpr IndexType TransformedFeatureDimensions = L1;
    static constexpr int       FC_0_OUTPUTS                 = L2;
    static constexpr int       FC_1_OUTPUTS                 = L3;

    Layers::AffineTransformSparseInput<TransformedFeatureDimensions, FC_0_OUTPUTS + 1> fc_0;
    Layers::SqrClippedReLU<FC_0_OUTPUTS + 1>                                           ac_sqr_0;
    Layers::ClippedReLU<FC_0_OUTPUTS + 1>                                              ac_0;
    Layers::AffineTransform<FC_0_OUTPUTS * 2, FC_1_OUTPUTS>                            fc_1;
    Layers::ClippedReLU<FC_1_OUTPUTS>                                                  ac_1;
    Layers::AffineTransform<FC_1_OUTPUTS, 1>                                           fc_2;

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t get_hash_value() {
        // input slice hash
        std::uint32_t hashValue = 0xEC42E90Du;
        hashValue ^= TransformedFeatureDimensions * 2;

        hashValue = decltype(fc_0)::get_hash_value(hashValue);
        hashValue = decltype(ac_0)::get_hash_value(hashValue);
        hashValue = decltype(fc_1)::get_hash_value(hashValue);
        hashValue = decltype(ac_1)::get_hash_value(hashValue);
        hashValue = decltype(fc_2)::get_hash_value(hashValue);

        return hashValue;
    }

    // Read network parameters
    bool read_parameters(std::istream& stream) {
        return fc_0.read_parameters(stream) && ac_0.read_parameters(stream)
            && fc_1.read_parameters(stream) && ac_1.read_parameters(stream)
            && fc_2.read_parameters(stream);
    }

    // Write network parameters
    bool write_parameters(std::ostream& stream) const {
        return fc_0.write_parameters(stream) && ac_0.write_parameters(stream)
            && fc_1.write_parameters(stream) && ac_1.write_parameters(stream)
            && fc_2.write_parameters(stream);
    }

    std::int32_t propagate(const TransformedFeatureType* transformedFeatures) {
        struct alignas(CacheLineSize) Buffer {
            alignas(CacheLineSize) typename decltype(fc_0)::OutputBuffer fc_0_out;
            alignas(CacheLineSize) typename decltype(ac_sqr_0)::OutputType
              ac_sqr_0_out[ceil_to_multiple<IndexType>(FC_0_OUTPUTS * 2, 32)];
            alignas(CacheLineSize) typename decltype(ac_0)::OutputBuffer ac_0_out;
            alignas(CacheLineSize) typename decltype(fc_1)::OutputBuffer fc_1_out;
            alignas(CacheLineSize) typename decltype(ac_1)::OutputBuffer ac_1_out;
            alignas(CacheLineSize) typename decltype(fc_2)::OutputBuffer fc_2_out;

            Buffer() { std::memset(this, 0, sizeof(*this)); }
        };

#if defined(__clang__) && (__APPLE__)
        // workaround for a bug reported with xcode 12
        static thread_local auto tlsBuffer = std::make_unique<Buffer>();
        // Access TLS only once, cache result.
        Buffer& buffer = *tlsBuffer;
#else
        alignas(CacheLineSize) static thread_local Buffer buffer;
#endif

        fc_0.propagate(transformedFeatures, buffer.fc_0_out);
        ac_sqr_0.propagate(buffer.fc_0_out, buffer.ac_sqr_0_out);
        ac_0.propagate(buffer.fc_0_out, buffer.ac_0_out);
        std::memcpy(buffer.ac_sqr_0_out + FC_0_OUTPUTS, buffer.ac_0_out,
                    FC_0_OUTPUTS * sizeof(typename decltype(ac_0)::OutputType));
        fc_1.propagate(buffer.ac_sqr_0_out, buffer.fc_1_out);
        ac_1.propagate(buffer.fc_1_out, buffer.ac_1_out);
        fc_2.propagate(buffer.ac_1_out, buffer.fc_2_out);

        // buffer.fc_0_out[FC_0_OUTPUTS] is such that 1.0 is equal to 127*(1<<WeightScaleBits) in
        // quantized form, but we want 1.0 to be equal to 600*OutputScale
        std::int32_t fwdOut =
          (buffer.fc_0_out[FC_0_OUTPUTS]) * (600 * OutputScale) / (127 * (1 << WeightScaleBits));
        std::int32_t outputValue = buffer.fc_2_out[0] + fwdOut;

        return outputValue;
    }
};

}  // namespace Stockfish::Eval::NNUE

#endif  // #ifndef NNUE_ARCHITECTURE_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Class for difference calculation of NNUE evaluation function

#ifndef NNUE_ACCUMULATOR_H_INCLUDED
#define NNUE_ACCUMULATOR_H_INCLUDED

namespace Stockfish::Eval::NNUE {

using BiasType       = std::int16_t;
using PSQTWeightType = std::int32_t;
using IndexType      = std::uint32_t;

// Class that holds the result of affine transformation of input features
template<IndexType Size>
struct alignas(CacheLineSize) Accumulator {
    std::int16_t accumulation[COLOR_NB][Size];
    std::int32_t psqtAccumulation[COLOR_NB][PSQTBuckets];
    bool         computed[COLOR_NB];
};

// AccumulatorCaches struct provides per-thread accumulator caches, where each
// cache contains multiple entries for each of the possible king squares.
// When the accumulator needs to be refreshed, the cached entry is used to more
// efficiently update the accumulator, instead of rebuilding it from scratch.
// This idea, was first described by Luecx (author of Koivisto) and
// is commonly referred to as "Finny Tables".
struct AccumulatorCaches {

    template<typename Networks>
    AccumulatorCaches(const Networks& networks) {
        clear(networks);
    }

    template<IndexType Size>
    struct alignas(CacheLineSize) Cache {

        struct alignas(CacheLineSize) Entry {
            BiasType       accumulation[Size];
            PSQTWeightType psqtAccumulation[PSQTBuckets];
            Bitboard       byColorBB[COLOR_NB];
            Bitboard       byTypeBB[PIECE_TYPE_NB];

            // To initialize a refresh entry, we set all its bitboards empty,
            // so we put the biases in the accumulation, without any weights on top
            void clear(const BiasType* biases) {

                std::memcpy(accumulation, biases, sizeof(accumulation));
                std::memset((uint8_t*) this + offsetof(Entry, psqtAccumulation), 0,
                            sizeof(Entry) - offsetof(Entry, psqtAccumulation));
            }
        };

        template<typename Network>
        void clear(const Network& network) {
            for (auto& entries1D : entries)
                for (auto& entry : entries1D)
                    entry.clear(network.featureTransformer->biases);
        }

        void clear(const BiasType* biases) {
            for (auto& entry : entries)
                entry.clear(biases);
        }

        std::array<Entry, COLOR_NB>& operator[](Square sq) { return entries[sq]; }

        std::array<std::array<Entry, COLOR_NB>, SQUARE_NB> entries;
    };

    template<typename Networks>
    void clear(const Networks& networks) {
        big.clear(networks.big);
        small.clear(networks.small);
    }

    Cache<TransformedFeatureDimensionsBig>   big;
    Cache<TransformedFeatureDimensionsSmall> small;
};

}  // namespace Stockfish::Eval::NNUE

#endif  // NNUE_ACCUMULATOR_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef BITBOARD_H_INCLUDED
#define BITBOARD_H_INCLUDED

namespace Stockfish {

namespace Bitboards {

void        init();
std::string pretty(Bitboard b);

}  // namespace Stockfish::Bitboards

constexpr Bitboard FileABB = 0x0101010101010101ULL;
constexpr Bitboard FileBBB = FileABB << 1;
constexpr Bitboard FileCBB = FileABB << 2;
constexpr Bitboard FileDBB = FileABB << 3;
constexpr Bitboard FileEBB = FileABB << 4;
constexpr Bitboard FileFBB = FileABB << 5;
constexpr Bitboard FileGBB = FileABB << 6;
constexpr Bitboard FileHBB = FileABB << 7;

constexpr Bitboard Rank1BB = 0xFF;
constexpr Bitboard Rank2BB = Rank1BB << (8 * 1);
constexpr Bitboard Rank3BB = Rank1BB << (8 * 2);
constexpr Bitboard Rank4BB = Rank1BB << (8 * 3);
constexpr Bitboard Rank5BB = Rank1BB << (8 * 4);
constexpr Bitboard Rank6BB = Rank1BB << (8 * 5);
constexpr Bitboard Rank7BB = Rank1BB << (8 * 6);
constexpr Bitboard Rank8BB = Rank1BB << (8 * 7);

extern uint8_t PopCnt16[1 << 16];
extern uint8_t SquareDistance[SQUARE_NB][SQUARE_NB];

extern Bitboard BetweenBB[SQUARE_NB][SQUARE_NB];
extern Bitboard LineBB[SQUARE_NB][SQUARE_NB];
extern Bitboard PseudoAttacks[PIECE_TYPE_NB][SQUARE_NB];
extern Bitboard PawnAttacks[COLOR_NB][SQUARE_NB];

// Magic holds all magic bitboards relevant data for a single square
struct Magic {
    Bitboard  mask;
    Bitboard  magic;
    Bitboard* attacks;
    unsigned  shift;

    // Compute the attack's index using the 'magic bitboards' approach
    unsigned index(Bitboard occupied) const {

        if (HasPext)
            return unsigned(pext(occupied, mask));

        if (Is64Bit)
            return unsigned(((occupied & mask) * magic) >> shift);

        unsigned lo = unsigned(occupied) & unsigned(mask);
        unsigned hi = unsigned(occupied >> 32) & unsigned(mask >> 32);
        return (lo * unsigned(magic) ^ hi * unsigned(magic >> 32)) >> shift;
    }
};

extern Magic RookMagics[SQUARE_NB];
extern Magic BishopMagics[SQUARE_NB];

constexpr Bitboard square_bb(Square s) {
    assert(is_ok(s));
    return (1ULL << s);
}

// Overloads of bitwise operators between a Bitboard and a Square for testing
// whether a given bit is set in a bitboard, and for setting and clearing bits.

inline Bitboard  operator&(Bitboard b, Square s) { return b & square_bb(s); }
inline Bitboard  operator|(Bitboard b, Square s) { return b | square_bb(s); }
inline Bitboard  operator^(Bitboard b, Square s) { return b ^ square_bb(s); }
inline Bitboard& operator|=(Bitboard& b, Square s) { return b |= square_bb(s); }
inline Bitboard& operator^=(Bitboard& b, Square s) { return b ^= square_bb(s); }

inline Bitboard operator&(Square s, Bitboard b) { return b & s; }
inline Bitboard operator|(Square s, Bitboard b) { return b | s; }
inline Bitboard operator^(Square s, Bitboard b) { return b ^ s; }

inline Bitboard operator|(Square s1, Square s2) { return square_bb(s1) | s2; }

constexpr bool more_than_one(Bitboard b) { return b & (b - 1); }

// rank_bb() and file_bb() return a bitboard representing all the squares on
// the given file or rank.

constexpr Bitboard rank_bb(Rank r) { return Rank1BB << (8 * r); }

constexpr Bitboard rank_bb(Square s) { return rank_bb(rank_of(s)); }

constexpr Bitboard file_bb(File f) { return FileABB << f; }

constexpr Bitboard file_bb(Square s) { return file_bb(file_of(s)); }

// Moves a bitboard one or two steps as specified by the direction D
template<Direction D>
constexpr Bitboard shift(Bitboard b) {
    return D == NORTH         ? b << 8
         : D == SOUTH         ? b >> 8
         : D == NORTH + NORTH ? b << 16
         : D == SOUTH + SOUTH ? b >> 16
         : D == EAST          ? (b & ~FileHBB) << 1
         : D == WEST          ? (b & ~FileABB) >> 1
         : D == NORTH_EAST    ? (b & ~FileHBB) << 9
         : D == NORTH_WEST    ? (b & ~FileABB) << 7
         : D == SOUTH_EAST    ? (b & ~FileHBB) >> 7
         : D == SOUTH_WEST    ? (b & ~FileABB) >> 9
                              : 0;
}

// Returns the squares attacked by pawns of the given color
// from the squares in the given bitboard.
template<Color C>
constexpr Bitboard pawn_attacks_bb(Bitboard b) {
    return C == WHITE ? shift<NORTH_WEST>(b) | shift<NORTH_EAST>(b)
                      : shift<SOUTH_WEST>(b) | shift<SOUTH_EAST>(b);
}

inline Bitboard pawn_attacks_bb(Color c, Square s) {

    assert(is_ok(s));
    return PawnAttacks[c][s];
}

// Returns a bitboard representing an entire line (from board edge
// to board edge) that intersects the two given squares. If the given squares
// are not on a same file/rank/diagonal, the function returns 0. For instance,
// line_bb(SQ_C4, SQ_F7) will return a bitboard with the A2-G8 diagonal.
inline Bitboard line_bb(Square s1, Square s2) {

    assert(is_ok(s1) && is_ok(s2));
    return LineBB[s1][s2];
}

// Returns a bitboard representing the squares in the semi-open
// segment between the squares s1 and s2 (excluding s1 but including s2). If the
// given squares are not on a same file/rank/diagonal, it returns s2. For instance,
// between_bb(SQ_C4, SQ_F7) will return a bitboard with squares D5, E6 and F7, but
// between_bb(SQ_E6, SQ_F8) will return a bitboard with the square F8. This trick
// allows to generate non-king evasion moves faster: the defending piece must either
// interpose itself to cover the check or capture the checking piece.
inline Bitboard between_bb(Square s1, Square s2) {

    assert(is_ok(s1) && is_ok(s2));
    return BetweenBB[s1][s2];
}

// Returns true if the squares s1, s2 and s3 are aligned either on a
// straight or on a diagonal line.
inline bool aligned(Square s1, Square s2, Square s3) { return line_bb(s1, s2) & s3; }

// distance() functions return the distance between x and y, defined as the
// number of steps for a king in x to reach y.

template<typename T1 = Square>
inline int distance(Square x, Square y);

template<>
inline int distance<File>(Square x, Square y) {
    return std::abs(file_of(x) - file_of(y));
}

template<>
inline int distance<Rank>(Square x, Square y) {
    return std::abs(rank_of(x) - rank_of(y));
}

template<>
inline int distance<Square>(Square x, Square y) {
    return SquareDistance[x][y];
}

inline int edge_distance(File f) { return std::min(f, File(FILE_H - f)); }

// Returns the pseudo attacks of the given piece type
// assuming an empty board.
template<PieceType Pt>
inline Bitboard attacks_bb(Square s) {

    assert((Pt != PAWN) && (is_ok(s)));
    return PseudoAttacks[Pt][s];
}

// Returns the attacks by the given piece
// assuming the board is occupied according to the passed Bitboard.
```

```cpp
// Sliding piece attacks do not continue passed an occupied square.
template<PieceType Pt>
inline Bitboard attacks_bb(Square s, Bitboard occupied) {

    assert((Pt != PAWN) && (is_ok(s)));

    switch (Pt)
    {
    case BISHOP :
        return BishopMagics[s].attacks[BishopMagics[s].index(occupied)];
    case ROOK :
        return RookMagics[s].attacks[RookMagics[s].index(occupied)];
    case QUEEN :
        return attacks_bb<BISHOP>(s, occupied) | attacks_bb<ROOK>(s, occupied);
    default :
        return PseudoAttacks[Pt][s];
    }
}

// Returns the attacks by the given piece
// assuming the board is occupied according to the passed Bitboard.
// Sliding piece attacks do not continue passed an occupied square.
inline Bitboard attacks_bb(PieceType pt, Square s, Bitboard occupied) {

    assert((pt != PAWN) && (is_ok(s)));

    switch (pt)
    {
    case BISHOP :
        return attacks_bb<BISHOP>(s, occupied);
    case ROOK :
        return attacks_bb<ROOK>(s, occupied);
    case QUEEN :
        return attacks_bb<BISHOP>(s, occupied) | attacks_bb<ROOK>(s, occupied);
    default :
        return PseudoAttacks[pt][s];
    }
}

// Counts the number of non-zero bits in a bitboard.
inline int popcount(Bitboard b) {

#ifndef USE_POPCNT

    union {
        Bitboard bb;
        uint16_t u[4];
    } v = {b};
    return PopCnt16[v.u[0]] + PopCnt16[v.u[1]] + PopCnt16[v.u[2]] + PopCnt16[v.u[3]];

#elif defined(_MSC_VER)

    return int(_mm_popcnt_u64(b));

#else  // Assumed gcc or compatible compiler

    return __builtin_popcountll(b);

#endif
}

// Returns the least significant bit in a non-zero bitboard.
inline Square lsb(Bitboard b) {
    assert(b);

#if defined(__GNUC__)  // GCC, Clang, ICX

    return Square(__builtin_ctzll(b));

#elif defined(_MSC_VER)
    #ifdef _WIN64  // MSVC, WIN64

    unsigned long idx;
    _BitScanForward64(&idx, b);
    return Square(idx);

    #else  // MSVC, WIN32
    unsigned long idx;

    if (b & 0xffffffff)
    {
        _BitScanForward(&idx, int32_t(b));
        return Square(idx);
    }
    else
    {
        _BitScanForward(&idx, int32_t(b >> 32));
        return Square(idx + 32);
    }
    #endif
#else  // Compiler is neither GCC nor MSVC compatible
    #error "Compiler not supported."
#endif
}

// Returns the most significant bit in a non-zero bitboard.
inline Square msb(Bitboard b) {
    assert(b);

#if defined(__GNUC__)  // GCC, Clang, ICX

    return Square(63 ^ __builtin_clzll(b));

#elif defined(_MSC_VER)
    #ifdef _WIN64  // MSVC, WIN64

    unsigned long idx;
    _BitScanReverse64(&idx, b);
    return Square(idx);

    #else  // MSVC, WIN32

    unsigned long idx;

    if (b >> 32)
    {
        _BitScanReverse(&idx, int32_t(b >> 32));
        return Square(idx + 32);
    }
    else
    {
        _BitScanReverse(&idx, int32_t(b));
        return Square(idx);
    }
    #endif
#else  // Compiler is neither GCC nor MSVC compatible
    #error "Compiler not supported."
#endif
}

// Returns the bitboard of the least significant
// square of a non-zero bitboard. It is equivalent to square_bb(lsb(bb)).
inline Bitboard least_significant_square_bb(Bitboard b) {
    assert(b);
    return b & -b;
}

// Finds and clears the least significant bit in a non-zero bitboard.
inline Square pop_lsb(Bitboard& b) {
    assert(b);
    const Square s = lsb(b);
    b &= b - 1;
    return s;
}

}  // namespace Stockfish

#endif  // #ifndef BITBOARD_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef POSITION_H_INCLUDED
#define POSITION_H_INCLUDED

namespace Stockfish {

class TranspositionTable;

// StateInfo struct stores information needed to restore a Position object to
// its previous state when we retract a move. Whenever a move is made on the
// board (by calling Position::do_move), a StateInfo object must be passed.

struct StateInfo {

    // Copied when making a move
    Key    materialKey;
    Key    pawnKey;
    Value  nonPawnMaterial[COLOR_NB];
    int    castlingRights;
    int    rule50;
    int    pliesFromNull;
    Square epSquare;

    // Not copied when making a move (will be recomputed anyhow)
    Key        key;
    Bitboard   checkersBB;
    StateInfo* previous;
    Bitboard   blockersForKing[COLOR_NB];
    Bitboard   pinners[COLOR_NB];
    Bitboard   checkSquares[PIECE_TYPE_NB];
    Piece      capturedPiece;
    int        repetition;

    // Used by NNUE
    Eval::NNUE::Accumulator<Eval::NNUE::TransformedFeatureDimensionsBig>   accumulatorBig;
    Eval::NNUE::Accumulator<Eval::NNUE::TransformedFeatureDimensionsSmall> accumulatorSmall;
    DirtyPiece                                                             dirtyPiece;
};

// A list to keep track of the position states along the setup moves (from the
// start position to the position just before the search starts). Needed by
// 'draw by repetition' detection. Use a std::deque because pointers to
// elements are not invalidated upon list resizing.
using StateListPtr = std::unique_ptr<std::deque<StateInfo>>;

// Position class stores information regarding the board representation as
// pieces, side to move, hash keys, castling info, etc. Important methods are
// do_move() and undo_move(), used by the search to update node info when
// traversing the search tree.
class Position {
   public:
    static void init();

    Position()                            = default;
    Position(const Position&)             = delete;
    Position& operator=(const Position&) = delete;

    // FEN string input/output
    Position&   set(const std::string& fenStr, bool isChess960, StateInfo* si);
    Position&   set(const std::string& code, Color c, StateInfo* si);
    std::string fen() const;

    // Position representation
    Bitboard pieces(PieceType pt = ALL_PIECES) const;
    template<typename... PieceTypes>
    Bitboard pieces(PieceType pt, PieceTypes... pts) const;
    Bitboard pieces(Color c) const;
    template<typename... PieceTypes>
    Bitboard pieces(Color c, PieceTypes... pts) const;
    Piece    piece_on(Square s) const;
    Square   ep_square() const;
    bool     empty(Square s) const;
    template<PieceType Pt>
    int count(Color c) const;
    template<PieceType Pt>
    int count() const;
    template<PieceType Pt>
    Square square(Color c) const;

    // Castling
    CastlingRights castling_rights(Color c) const;
    bool           can_castle(CastlingRights cr) const;
    bool           castling_impeded(CastlingRights cr) const;
    Square         castling_rook_square(CastlingRights cr) const;

    // Checking
    Bitboard checkers() const;
    Bitboard blockers_for_king(Color c) const;
    Bitboard check_squares(PieceType pt) const;
    Bitboard pinners(Color c) const;

    // Attacks to/from a given square
    Bitboard attackers_to(Square s) const;
    Bitboard attackers_to(Square s, Bitboard occupied) const;
    void     update_slider_blockers(Color c) const;
    template<PieceType Pt>
    Bitboard attacks_by(Color c) const;

    // Properties of moves
    bool  legal(Move m) const;
    bool  pseudo_legal(const Move m) const;
    bool  capture(Move m) const;
    bool  capture_stage(Move m) const;
    bool  gives_check(Move m) const;
    Piece moved_piece(Move m) const;
    Piece captured_piece() const;

    // Doing and undoing moves
    void do_move(Move m, StateInfo& newSt);
    void do_move(Move m, StateInfo& newSt, bool givesCheck);
    void undo_move(Move m);
    void do_null_move(StateInfo& newSt, TranspositionTable& tt);
    void undo_null_move();

    // Static Exchange Evaluation
    bool see_ge(Move m, int threshold = 0) const;

    // Accessing hash keys
    Key key() const;
    Key key_after(Move m) const;
    Key material_key() const;
    Key pawn_key() const;

    // Other properties of the position
    Color side_to_move() const;
    int   game_ply() const;
    bool  is_chess960() const;
    bool  is_draw(int ply) const;
    bool  upcoming_repetition(int ply) const;
    bool  has_repeated() const;
    int   rule50_count() const;
    Value non_pawn_material(Color c) const;
    Value non_pawn_material() const;

    // Position consistency check, for debugging
    bool pos_is_ok() const;
    void flip();

    // Used by NNUE
    StateInfo* state() const;

    void put_piece(Piece pc, Square s);
    void remove_piece(Square s);

   private:
    // Initialization helpers (used while setting up a position)
    void set_castling_right(Color c, Square rfrom);
    void set_state() const;
    void set_check_info() const;

    // Other helpers
    void move_piece(Square from, Square to);
    template<bool Do>
    void do_castling(Color us, Square from, Square& to, Square& rfrom, Square& rto);
    template<bool AfterMove>
    Key adjust_key50(Key k) const;

    // Data members
    Piece      board[SQUARE_NB];
    Bitboard   byTypeBB[PIECE_TYPE_NB];
    Bitboard   byColorBB[COLOR_NB];
    int        pieceCount[PIECE_NB];
    int        castlingRightsMask[SQUARE_NB];
    Square     castlingRookSquare[CASTLING_RIGHT_NB];
    Bitboard   castlingPath[CASTLING_RIGHT_NB];
    StateInfo* st;
    int        gamePly;
    Color      sideToMove;
    bool       chess960;
};

std::ostream& operator<<(std::ostream& os, const Position& pos);

inline Color Position::side_to_move() const { return sideToMove; }

inline Piece Position::piece_on(Square s) const {
    assert(is_ok(s));
    return board[s];
}

inline bool Position::empty(Square s) const { return piece_on(s) == NO_PIECE; }

inline Piece Position::moved_piece(Move m) const { return piece_on(m.from_sq()); }

inline Bitboard Position::pieces(PieceType pt) const { return byTypeBB[pt]; }

template<typename... PieceTypes>
inline Bitboard Position::pieces(PieceType pt, PieceTypes... pts) const {
    return pieces(pt) | pieces(pts...);
}

inline Bitboard Position::pieces(Color c) const { return byColorBB[c]; }

template<typename... PieceTypes>
inline Bitboard Position::pieces(Color c, PieceTypes... pts) const {
    return pieces(c) & pieces(pts...);
}

template<PieceType Pt>
inline int Position::count(Color c) const {
    return pieceCount[make_piece(c, Pt)];
}

template<PieceType Pt>
inline int Position::count() const {
    return count<Pt>(WHITE) + count<Pt>(BLACK);
}

template<PieceType Pt>
inline Square Position::square(Color c) const {
    assert(count<Pt>(c) == 1);
    return lsb(pieces(c, Pt));
}

inline Square Position::ep_square() const { return st->epSquare; }

inline bool Position::can_castle(CastlingRights cr) const { return st->castlingRights & cr; }

inline CastlingRights Position::castling_rights(Color c) const {
    return c & CastlingRights(st->castlingRights);
}

inline bool Position::castling_impeded(CastlingRights cr) const {
    assert(cr == WHITE_OO || cr == WHITE_OOO || cr == BLACK_OO || cr == BLACK_OOO);
    return pieces() & castlingPath[cr];
}

inline Square Position::castling_rook_square(CastlingRights cr) const {
    assert(cr == WHITE_OO || cr == WHITE_OOO || cr == BLACK_OO || cr == BLACK_OOO);
    return castlingRookSquare[cr];
}

inline Bitboard Position::attackers_to(Square s) const { return attackers_to(s, pieces()); }

template<PieceType Pt>
inline Bitboard Position::attacks_by(Color c) const {

    if constexpr (Pt == PAWN)
        return c == WHITE ? pawn_attacks_bb<WHITE>(pieces(WHITE, PAWN))
                          : pawn_attacks_bb<BLACK>(pieces(BLACK, PAWN));
    else
    {
        Bitboard threats   = 0;
        Bitboard attackers = pieces(c, Pt);
        while (attackers)
            threats |= attacks_bb<Pt>(pop_lsb(attackers), pieces());
        return threats;
    }
}

inline Bitboard Position::checkers() const { return st->checkersBB; }

inline Bitboard Position::blockers_for_king(Color c) const { return st->blockersForKing[c]; }

inline Bitboard Position::pinners(Color c) const { return st->pinners[c]; }

inline Bitboard Position::check_squares(PieceType pt) const { return st->checkSquares[pt]; }

inline Key Position::key() const { return adjust_key50<false>(st->key); }

template<bool AfterMove>
inline Key Position::adjust_key50(Key k) const {
    return st->rule50 < 14 - AfterMove ? k : k ^ make_key((st->rule50 - (14 - AfterMove)) / 8);
}

inline Key Position::pawn_key() const { return st->pawnKey; }

inline Key Position::material_key() const { return st->materialKey; }

inline Value Position::non_pawn_material(Color c) const { return st->nonPawnMaterial[c]; }

inline Value Position::non_pawn_material() const {
    return non_pawn_material(WHITE) + non_pawn_material(BLACK);
}

inline int Position::game_ply() const { return gamePly; }

inline int Position::rule50_count() const { return st->rule50; }

inline bool Position::is_chess960() const { return chess960; }

inline bool Position::capture(Move m) const {
    assert(m.is_ok());
    return (!empty(m.to_sq()) && m.type_of() != CASTLING) || m.type_of() == EN_PASSANT;
}

// Returns true if a move is generated from the capture stage, having also
// queen promotions covered, i.e. consistency with the capture stage move
// generation is needed to avoid the generation of duplicate moves.
inline bool Position::capture_stage(Move m) const {
    assert(m.is_ok());
    return capture(m) || m.promotion_type() == QUEEN;
}

inline Piece Position::captured_piece() const { return st->capturedPiece; }

inline void Position::put_piece(Piece pc, Square s) {

    board[s] = pc;
    byTypeBB[ALL_PIECES] |= byTypeBB[type_of(pc)] |= s;
    byColorBB[color_of(pc)] |= s;
    pieceCount[pc]++;
    pieceCount[make_piece(color_of(pc), ALL_PIECES)]++;
}

inline void Position::remove_piece(Square s) {

    Piece pc = board[s];
    byTypeBB[ALL_PIECES] ^= s;
    byTypeBB[type_of(pc)] ^= s;
    byColorBB[color_of(pc)] ^= s;
    board[s] = NO_PIECE;
    pieceCount[pc]--;
    pieceCount[make_piece(color_of(pc), ALL_PIECES)]--;
}

inline void Position::move_piece(Square from, Square to) {

    Piece    pc     = board[from];
    Bitboard fromTo = from | to;
    byTypeBB[ALL_PIECES] ^= fromTo;
    byTypeBB[type_of(pc)] ^= fromTo;
    byColorBB[color_of(pc)] ^= fromTo;
    board[from] = NO_PIECE;
    board[to]   = pc;
}

inline void Position::do_move(Move m, StateInfo& newSt) { do_move(m, newSt, gives_check(m)); }

inline StateInfo* Position::state() const { return st; }

}  // namespace Stockfish

#endif  // #ifndef POSITION_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// A class that converts the input features of the NNUE evaluation function

#ifndef NNUE_FEATURE_TRANSFORMER_H_INCLUDED
#define NNUE_FEATURE_TRANSFORMER_H_INCLUDED

namespace Stockfish::Eval::NNUE {

using BiasType       = std::int16_t;
using WeightType     = std::int16_t;
using PSQTWeightType = std::int32_t;

// If vector instructions are enabled, we update and refresh the
// accumulator tile by tile such that each tile fits in the CPU's
// vector registers.
#define VECTOR

static_assert(PSQTBuckets % 8 == 0,
              "Per feature PSQT values cannot be processed at granularity lower than 8 at a time.");

#ifdef USE_AVX512
using vec_t      = __m512i;
using psqt_vec_t = __m256i;
    #define vec_load(a) _mm512_load_si512(a)
    #define vec_store(a, b) _mm512_store_si512(a, b)
    #define vec_add_16(a, b) _mm512_add_epi16(a, b)
    #define vec_sub_16(a, b) _mm512_sub_epi16(a, b)
    #define vec_mulhi_16(a, b) _mm512_mulhi_epi16(a, b)
    #define vec_zero() _mm512_setzero_epi32()
    #define vec_set_16(a) _mm512_set1_epi16(a)
    #define vec_max_16(a, b) _mm512_max_epi16(a, b)
    #define vec_min_16(a, b) _mm512_min_epi16(a, b)
    #define vec_slli_16(a, b) _mm512_slli_epi16(a, b)
    // Inverse permuted at load time
    #define vec_packus_16(a, b) _mm512_packus_epi16(a, b)
    #define vec_load_psqt(a) _mm256_load_si256(a)
    #define vec_store_psqt(a, b) _mm256_store_si256(a, b)
    #define vec_add_psqt_32(a, b) _mm256_add_epi32(a, b)
    #define vec_sub_psqt_32(a, b) _mm256_sub_epi32(a, b)
    #define vec_zero_psqt() _mm256_setzero_si256()
    #define NumRegistersSIMD 16
    #define MaxChunkSize 64

#elif USE_AVX2
using vec_t      = __m256i;
using psqt_vec_t = __m256i;
    #define vec_load(a) _mm256_load_si256(a)
    #define vec_store(a, b) _mm256_store_si256(a, b)
    #define vec_add_16(a, b) _mm256_add_epi16(a, b)
    #define vec_sub_16(a, b) _mm256_sub_epi16(a, b)
    #define vec_mulhi_16(a, b) _mm256_mulhi_epi16(a, b)
    #define vec_zero() _mm256_setzero_si256()
    #define vec_set_16(a) _mm256_set1_epi16(a)
    #define vec_max_16(a, b) _mm256_max_epi16(a, b)
    #define vec_min_16(a, b) _mm256_min_epi16(a, b)
    #define vec_slli_16(a, b) _mm256_slli_epi16(a, b)
    // Inverse permuted at load time
    #define vec_packus_16(a, b) _mm256_packus_epi16(a, b)
    #define vec_load_psqt(a) _mm256_load_si256(a)
    #define vec_store_psqt(a, b) _mm256_store_si256(a, b)
    #define vec_add_psqt_32(a, b) _mm256_add_epi32(a, b)
    #define vec_sub_psqt_32(a, b) _mm256_sub_epi32(a, b)
    #define vec_zero_psqt() _mm256_setzero_si256()
    #define NumRegistersSIMD 16
    #define MaxChunkSize 32

#elif USE_SSE2
using vec_t      = __m128i;
using psqt_vec_t = __m128i;
    #define vec_load(a) (*(a))
    #define vec_store(a, b) *(a) = (b)
    #define vec_add_16(a, b) _mm_add_epi16(a, b)
    #define vec_sub_16(a, b) _mm_sub_epi16(a, b)
    #define vec_mulhi_16(a, b) _mm_mulhi_epi16(a, b)
    #define vec_zero() _mm_setzero_si128()
    #define vec_set_16(a) _mm_set1_epi16(a)
    #define vec_max_16(a, b) _mm_max_epi16(a, b)
    #define vec_min_16(a, b) _mm_min_epi16(a, b)
    #define vec_slli_16(a, b) _mm_slli_epi16(a, b)
    #define vec_packus_16(a, b) _mm_packus_epi16(a, b)
    #define vec_load_psqt(a) (*(a))
    #define vec_store_psqt(a, b) *(a) = (b)
    #define vec_add_psqt_32(a, b) _mm_add_epi32(a, b)
    #define vec_sub_psqt_32(a, b) _mm_sub_epi32(a, b)
    #define vec_zero_psqt() _mm_setzero_si128()
    #define NumRegistersSIMD (Is64Bit ? 16 : 8)
    #define MaxChunkSize 16

#elif USE_NEON
using vec_t      = int16x8_t;
using psqt_vec_t = int32x4_t;
    #define vec_load(a) (*(a))
    #define vec_store(a, b) *(a) = (b)
    #define vec_add_16(a, b) vaddq_s16(a, b)
    #define vec_sub_16(a, b) vsubq_s16(a, b)
    #define vec_mulhi_16(a, b) vqdmulhq_s16(a, b)
    #define vec_zero() \
        vec_t { 0 }
    #define vec_set_16(a) vdupq_n_s16(a)
    #define vec_max_16(a, b) vmaxq_s16(a, b)
    #define vec_min_16(a, b) vminq_s16(a, b)
    #define vec_slli_16(a, b) vshlq_s16(a, vec_set_16(b))
    #define vec_packus_16(a, b) reinterpret_cast<vec_t>(vcombine_u8(vqmovun_s16(a), vqmovun_s16(b)))
    #define vec_load_psqt(a) (*(a))
    #define vec_store_psqt(a, b) *(a) = (b)
    #define vec_add_psqt_32(a, b) vaddq_s32(a, b)
    #define vec_sub_psqt_32(a, b) vsubq_s32(a, b)
    #define vec_zero_psqt() \
        psqt_vec_t { 0 }
    #define NumRegistersSIMD 16
    #define MaxChunkSize 16

#else
    #undef VECTOR

#endif

#ifdef VECTOR

    // Compute optimal SIMD register count for feature transformer accumulation.

    // We use __m* types as template arguments, which causes GCC to emit warnings
    // about losing some attribute information. This is irrelevant to us as we
    // only take their size, so the following pragma are harmless.
    #if defined(__GNUC__)
        #pragma GCC diagnostic push
        #pragma GCC diagnostic ignored "-Wignored-attributes"
    #endif

template<typename SIMDRegisterType, typename LaneType, int NumLanes, int MaxRegisters>
static constexpr int BestRegisterCount() {
    #define RegisterSize sizeof(SIMDRegisterType)
    #define LaneSize sizeof(LaneType)

    static_assert(RegisterSize >= LaneSize);
    static_assert(MaxRegisters <= NumRegistersSIMD);
    static_assert(MaxRegisters > 0);
    static_assert(NumRegistersSIMD > 0);
    static_assert(RegisterSize % LaneSize == 0);
    static_assert((NumLanes * LaneSize) % RegisterSize == 0);

    const int ideal = (NumLanes * LaneSize) / RegisterSize;
    if (ideal <= MaxRegisters)
        return ideal;

    // Look for the largest divisor of the ideal register count that is smaller than MaxRegisters
    for (int divisor = MaxRegisters; divisor > 1; --divisor)
        if (ideal % divisor == 0)
            return divisor;

    return 1;
}
    #if defined(__GNUC__)
        #pragma GCC diagnostic pop
    #endif
#endif

// Input feature converter
template<IndexType                                 TransformedFeatureDimensions,
         Accumulator<TransformedFeatureDimensions> StateInfo::*accPtr>
class FeatureTransformer {

    // Number of output dimensions for one side
    static constexpr IndexType HalfDimensions = TransformedFeatureDimensions;

   private:
#ifdef VECTOR
    static constexpr int NumRegs =
      BestRegisterCount<vec_t, WeightType, TransformedFeatureDimensions, NumRegistersSIMD>();
    static constexpr int NumPsqtRegs =
      BestRegisterCount<psqt_vec_t, PSQTWeightType, PSQTBuckets, NumRegistersSIMD>();

    static constexpr IndexType TileHeight     = NumRegs * sizeof(vec_t) / 2;
    static constexpr IndexType PsqtTileHeight = NumPsqtRegs * sizeof(psqt_vec_t) / 4;
    static_assert(HalfDimensions % TileHeight == 0, "TileHeight must divide HalfDimensions");
    static_assert(PSQTBuckets % PsqtTileHeight == 0, "PsqtTileHeight must divide PSQTBuckets");
#endif

   public:
    // Output type
    using OutputType = TransformedFeatureType;

    // Number of input/output dimensions
    static constexpr IndexType InputDimensions  = FeatureSet::Dimensions;
    static constexpr IndexType OutputDimensions = HalfDimensions;

    // Size of forward propagation buffer
    static constexpr std::size_t BufferSize = OutputDimensions * sizeof(OutputType);

    // Hash value embedded in the evaluation file
    static constexpr std::uint32_t get_hash_value() {
        return FeatureSet::HashValue ^ (OutputDimensions * 2);
    }

    static constexpr void order_packs([[maybe_unused]] uint64_t* v) {
#if defined(USE_AVX512)  // _mm512_packs_epi16 ordering
        uint64_t tmp0 = v[2], tmp1 = v[3];
        v[2] = v[8], v[3] = v[9];
        v[8] = v[4], v[9] = v[5];
        v[4] = tmp0, v[5] = tmp1;
        tmp0 = v[6], tmp1 = v[7];
        v[6] = v[10], v[7] = v[11];
        v[10] = v[12], v[11] = v[13];
        v[12] = tmp0, v[13] = tmp1;
#elif defined(USE_AVX2)  // _mm256_packs_epi16 ordering
        std::swap(v[2], v[4]);
        std::swap(v[3], v[5]);
#endif
    }

    static constexpr void inverse_order_packs([[maybe_unused]] uint64_t* v) {
#if defined(USE_AVX512)  // Inverse _mm512_packs_epi16 ordering
        uint64_t tmp0 = v[2], tmp1 = v[3];
        v[2] = v[4], v[3] = v[5];
        v[4] = v[8], v[5] = v[9];
        v[8] = tmp0, v[9] = tmp1;
        tmp0 = v[6], tmp1 = v[7];
        v[6] = v[12], v[7] = v[13];
        v[12] = v[10], v[13] = v[11];
        v[10] = tmp0, v[11] = tmp1;
#elif defined(USE_AVX2)  // Inverse _mm256_packs_epi16 ordering
        std::swap(v[2], v[4]);
        std::swap(v[3], v[5]);
#endif
    }

    void permute_weights([[maybe_unused]] void (*order_fn)(uint64_t*)) const {
#if defined(USE_AVX2)
    #if defined(USE_AVX512)
        constexpr IndexType di = 16;
    #else
        constexpr IndexType di = 8;
    #endif
        uint64_t* b = reinterpret_cast<uint64_t*>(const_cast<BiasType*>(&biases[0]));
        for (IndexType i = 0; i < HalfDimensions * sizeof(BiasType) / sizeof(uint64_t); i += di)
            order_fn(&b[i]);

        for (IndexType j = 0; j < InputDimensions; ++j)
        {
            uint64_t* w =
              reinterpret_cast<uint64_t*>(const_cast<WeightType*>(&weights[j * HalfDimensions]));
            for (IndexType i = 0; i < HalfDimensions * sizeof(WeightType) / sizeof(uint64_t);
                 i += di)
                order_fn(&w[i]);
        }
#endif
    }

    inline void scale_weights(bool read) const {
        for (IndexType j = 0; j < InputDimensions; ++j)
        {
            WeightType* w = const_cast<WeightType*>(&weights[j * HalfDimensions]);
            for (IndexType i = 0; i < HalfDimensions; ++i)
                w[i] = read ? w[i] * 2 : w[i] / 2;
        }

        BiasType* b = const_cast<BiasType*>(biases);
        for (IndexType i = 0; i < HalfDimensions; ++i)
            b[i] = read ? b[i] * 2 : b[i] / 2;
    }

    // Read network parameters
    bool read_parameters(std::istream& stream) {

        read_leb_128<BiasType>(stream, biases, HalfDimensions);
        read_leb_128<WeightType>(stream, weights, HalfDimensions * InputDimensions);
        read_leb_128<PSQTWeightType>(stream, psqtWeights, PSQTBuckets * InputDimensions);

        permute_weights(inverse_order_packs);
        scale_weights(true);
        return !stream.fail();
    }

    // Write network parameters
    bool write_parameters(std::ostream& stream) const {

        permute_weights(order_packs);
        scale_weights(false);

        write_leb_128<BiasType>(stream, biases, HalfDimensions);
        write_leb_128<WeightType>(stream, weights, HalfDimensions * InputDimensions);
        write_leb_128<PSQTWeightType>(stream, psqtWeights, PSQTBuckets * InputDimensions);

        permute_weights(inverse_order_packs);
        scale_weights(true);
        return !stream.fail();
    }

    // Convert input features
    std::int32_t transform(const Position&                           pos,
                           AccumulatorCaches::Cache<HalfDimensions>* cache,
                           OutputType*                               output,
                           int                                       bucket) const {
        update_accumulator<WHITE>(pos, cache);
        update_accumulator<BLACK>(pos, cache);

        const Color perspectives[2]  = {pos.side_to_move(), ~pos.side_to_move()};
        const auto& psqtAccumulation = (pos.state()->*accPtr).psqtAccumulation;
        const auto  psqt =
          (psqtAccumulation[perspectives[0]][bucket] - psqtAccumulation[perspectives[1]][bucket])
          / 2;

        const auto& accumulation = (pos.state()->*accPtr).accumulation;

        for (IndexType p = 0; p < 2; ++p)
        {
            const IndexType offset = (HalfDimensions / 2) * p;

#if defined(VECTOR)

            constexpr IndexType OutputChunkSize = MaxChunkSize;
            static_assert((HalfDimensions / 2) % OutputChunkSize == 0);
            constexpr IndexType NumOutputChunks = HalfDimensions / 2 / OutputChunkSize;

            const vec_t Zero = vec_zero();
            const vec_t One  = vec_set_16(127 * 2);

            const vec_t* in0 = reinterpret_cast<const vec_t*>(&(accumulation[perspectives[p]][0]));
            const vec_t* in1 =
              reinterpret_cast<const vec_t*>(&(accumulation[perspectives[p]][HalfDimensions / 2]));
            vec_t* out = reinterpret_cast<vec_t*>(output + offset);

            // Per the NNUE architecture, here we want to multiply pairs of
            // clipped elements and divide the product by 128. To do this,
            // we can naively perform min/max operation to clip each of the
            // four int16 vectors, mullo pairs together, then pack them into
            // one int8 vector. However, there exists a faster way.

            // The idea here is to use the implicit clipping from packus to
            // save us two vec_max_16 instructions. This clipping works due
            // to the fact that any int16 integer below zero will be zeroed
            // on packus.

            // Consider the case where the second element is negative.
            // If we do standard clipping, that element will be zero, which
            // means our pairwise product is zero. If we perform packus and
            // remove the lower-side clip for the second element, then our
            // product before packus will be negative, and is zeroed on pack.
            // The two operation produce equivalent results, but the second
            // one (using packus) saves one max operation per pair.

            // But here we run into a problem: mullo does not preserve the
            // sign of the multiplication. We can get around this by doing
            // mulhi, which keeps the sign. But that requires an additional
            // tweak.

            // mulhi cuts off the last 16 bits of the resulting product,
            // which is the same as performing a rightward shift of 16 bits.
            // We can use this to our advantage. Recall that we want to
            // divide the final product by 128, which is equivalent to a
            // 7-bit right shift. Intuitively, if we shift the clipped
            // value left by 9, and perform mulhi, which shifts the product
            // right by 16 bits, then we will net a right shift of 7 bits.
            // However, this won't work as intended. Since we clip the
            // values to have a maximum value of 127, shifting it by 9 bits
            // might occupy the signed bit, resulting in some positive
            // values being interpreted as negative after the shift.

            // There is a way, however, to get around this limitation. When
            // loading the network, scale accumulator weights and biases by
            // 2. To get the same pairwise multiplication result as before,
            // we need to divide the product by 128 * 2 * 2 = 512, which
            // amounts to a right shift of 9 bits. So now we only have to
            // shift left by 7 bits, perform mulhi (shifts right by 16 bits)
            // and net a 9 bit right shift. Since we scaled everything by
            // two, the values are clipped at 127 * 2 = 254, which occupies
            // 8 bits. Shifting it by 7 bits left will no longer occupy the
            // signed bit, so we are safe.

            // Note that on NEON processors, we shift left by 6 instead
            // because the instruction "vqdmulhq_s16" also doubles the
            // return value after the multiplication, adding an extra shift
            // to the left by 1, so we compensate by shifting less before
            // the multiplication.

            constexpr int shift =
    #if defined(USE_SSE2)
              7;
    #else
              6;
    #endif

            for (IndexType j = 0; j < NumOutputChunks; ++j)
            {
                const vec_t sum0a =
                  vec_slli_16(vec_max_16(vec_min_16(in0[j * 2 + 0], One), Zero), shift);
                const vec_t sum0b =
                  vec_slli_16(vec_max_16(vec_min_16(in0[j * 2 + 1], One), Zero), shift);
                const vec_t sum1a = vec_min_16(in1[j * 2 + 0], One);
                const vec_t sum1b = vec_min_16(in1[j * 2 + 1], One);

                const vec_t pa = vec_mulhi_16(sum0a, sum1a);
                const vec_t pb = vec_mulhi_16(sum0b, sum1b);

                out[j] = vec_packus_16(pa, pb);
            }

#else

            for (IndexType j = 0; j < HalfDimensions / 2; ++j)
            {
                BiasType sum0 = accumulation[static_cast<int>(perspectives[p])][j + 0];
                BiasType sum1 =
                  accumulation[static_cast<int>(perspectives[p])][j + HalfDimensions / 2];
                sum0               = std::clamp<BiasType>(sum0, 0, 127 * 2);
                sum1               = std::clamp<BiasType>(sum1, 0, 127 * 2);
                output[offset + j] = static_cast<OutputType>(unsigned(sum0 * sum1) / 512);
            }

#endif
        }

        return psqt;
    }  // end of function transform()

    void hint_common_access(const Position&                           pos,
                            AccumulatorCaches::Cache<HalfDimensions>* cache) const {
        hint_common_access_for_perspective<WHITE>(pos, cache);
        hint_common_access_for_perspective<BLACK>(pos, cache);
    }

   private:
    template<Color Perspective>
    [[nodiscard]] std::pair<StateInfo*, StateInfo*>
    try_find_computed_accumulator(const Position& pos) const {
        // Look for a usable accumulator of an earlier position. We keep track
        // of the estimated gain in terms of features to be added/subtracted.
        StateInfo *st = pos.state(), *next = nullptr;
        int        gain = FeatureSet::refresh_cost(pos);
        while (st->previous && !(st->*accPtr).computed[Perspective])
        {
            // This governs when a full feature refresh is needed and how many
            // updates are better than just one full refresh.
            if (FeatureSet::requires_refresh(st, Perspective)
                || (gain -= FeatureSet::update_cost(st) + 1) < 0)
                break;
            next = st;
            st   = st->previous;
        }
        return {st, next};
    }

    // NOTE: The parameter states_to_update is an array of position states.
    //       All states must be sequential, that is states_to_update[i] must
    //       either be reachable by repeatedly applying ->previous from
    //       states_to_update[i+1], and computed_st must be reachable by
    //       repeatedly applying ->previous on states_to_update[0].
    template<Color Perspective, size_t N>
    void update_accumulator_incremental(const Position& pos,
                                        StateInfo*      computed_st,
                                        StateInfo*      states_to_update[N]) const {
        static_assert(N > 0);
        assert([&]() {
            for (size_t i = 0; i < N; ++i)
            {
                if (states_to_update[i] == nullptr)
                    return false;
            }
            return true;
        }());

#ifdef VECTOR
        // Gcc-10.2 unnecessarily spills AVX2 registers if this array
        // is defined in the VECTOR code below, once in each branch.
        vec_t      acc[NumRegs];
        psqt_vec_t psqt[NumPsqtRegs];
#endif

        // Update incrementally going back through states_to_update.
        // Gather all features to be updated.
        const Square ksq = pos.square<KING>(Perspective);

        // The size must be enough to contain the largest possible update.
        // That might depend on the feature set and generally relies on the
        // feature set's update cost calculation to be correct and never allow
        // updates with more added/removed features than MaxActiveDimensions.
        FeatureSet::IndexList removed[N], added[N];

        for (int i = N - 1; i >= 0; --i)
        {
            (states_to_update[i]->*accPtr).computed[Perspective] = true;

            const StateInfo* end_state = i == 0 ? computed_st : states_to_update[i - 1];

            for (StateInfo* st2 = states_to_update[i]; st2 != end_state; st2 = st2->previous)
                FeatureSet::append_changed_indices<Perspective>(ksq, st2->dirtyPiece, removed[i],
                                                                added[i]);
        }

        StateInfo* st = computed_st;

        // Now update the accumulators listed in states_to_update[],
        // where the last element is a sentinel.
#ifdef VECTOR

        if (N == 1 && (removed[0].size() == 1 || removed[0].size() == 2) && added[0].size() == 1)
        {
            assert(states_to_update[0]);

            auto accIn =
              reinterpret_cast<const vec_t*>(&(st->*accPtr).accumulation[Perspective][0]);
            auto accOut = reinterpret_cast<vec_t*>(
              &(states_to_update[0]->*accPtr).accumulation[Perspective][0]);

            const IndexType offsetR0 = HalfDimensions * removed[0][0];
            auto            columnR0 = reinterpret_cast<const vec_t*>(&weights[offsetR0]);
            const IndexType offsetA  = HalfDimensions * added[0][0];
            auto            columnA  = reinterpret_cast<const vec_t*>(&weights[offsetA]);

            if (removed[0].size() == 1)
            {
                for (IndexType k = 0; k < HalfDimensions * sizeof(std::int16_t) / sizeof(vec_t);
                     ++k)
                    accOut[k] = vec_add_16(vec_sub_16(accIn[k], columnR0[k]), columnA[k]);
            }
            else
            {
                const IndexType offsetR1 = HalfDimensions * removed[0][1];
                auto            columnR1 = reinterpret_cast<const vec_t*>(&weights[offsetR1]);

                for (IndexType k = 0; k < HalfDimensions * sizeof(std::int16_t) / sizeof(vec_t);
                     ++k)
                    accOut[k] = vec_sub_16(vec_add_16(accIn[k], columnA[k]),
                                           vec_add_16(columnR0[k], columnR1[k]));
            }

            auto accPsqtIn =
              reinterpret_cast<const psqt_vec_t*>(&(st->*accPtr).psqtAccumulation[Perspective][0]);
            auto accPsqtOut = reinterpret_cast<psqt_vec_t*>(
              &(states_to_update[0]->*accPtr).psqtAccumulation[Perspective][0]);

            const IndexType offsetPsqtR0 = PSQTBuckets * removed[0][0];
            auto columnPsqtR0 = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offsetPsqtR0]);
            const IndexType offsetPsqtA = PSQTBuckets * added[0][0];
            auto columnPsqtA = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offsetPsqtA]);

            if (removed[0].size() == 1)
            {
                for (std::size_t k = 0; k < PSQTBuckets * sizeof(std::int32_t) / sizeof(psqt_vec_t);
                     ++k)
                    accPsqtOut[k] = vec_add_psqt_32(vec_sub_psqt_32(accPsqtIn[k], columnPsqtR0[k]),
                                                    columnPsqtA[k]);
            }
            else
            {
                const IndexType offsetPsqtR1 = PSQTBuckets * removed[0][1];
                auto columnPsqtR1 = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offsetPsqtR1]);

                for (std::size_t k = 0; k < PSQTBuckets * sizeof(std::int32_t) / sizeof(psqt_vec_t);
                     ++k)
                    accPsqtOut[k] =
                      vec_sub_psqt_32(vec_add_psqt_32(accPsqtIn[k], columnPsqtA[k]),
                                      vec_add_psqt_32(columnPsqtR0[k], columnPsqtR1[k]));
            }
        }
        else
        {
            for (IndexType j = 0; j < HalfDimensions / TileHeight; ++j)
            {
                // Load accumulator
                auto accTileIn = reinterpret_cast<const vec_t*>(
                  &(st->*accPtr).accumulation[Perspective][j * TileHeight]);
                for (IndexType k = 0; k < NumRegs; ++k)
                    acc[k] = vec_load(&accTileIn[k]);

                for (IndexType i = 0; i < N; ++i)
                {
                    // Difference calculation for the deactivated features
                    for (const auto index : removed[i])
                    {
                        const IndexType offset = HalfDimensions * index + j * TileHeight;
                        auto            column = reinterpret_cast<const vec_t*>(&weights[offset]);
                        for (IndexType k = 0; k < NumRegs; ++k)
                            acc[k] = vec_sub_16(acc[k], column[k]);
                    }

                    // Difference calculation for the activated features
                    for (const auto index : added[i])
                    {
                        const IndexType offset = HalfDimensions * index + j * TileHeight;
                        auto            column = reinterpret_cast<const vec_t*>(&weights[offset]);
                        for (IndexType k = 0; k < NumRegs; ++k)
                            acc[k] = vec_add_16(acc[k], column[k]);
                    }

                    // Store accumulator
                    auto accTileOut = reinterpret_cast<vec_t*>(
                      &(states_to_update[i]->*accPtr).accumulation[Perspective][j * TileHeight]);
                    for (IndexType k = 0; k < NumRegs; ++k)
                        vec_store(&accTileOut[k], acc[k]);
                }
            }

            for (IndexType j = 0; j < PSQTBuckets / PsqtTileHeight; ++j)
            {
                // Load accumulator
                auto accTilePsqtIn = reinterpret_cast<const psqt_vec_t*>(
                  &(st->*accPtr).psqtAccumulation[Perspective][j * PsqtTileHeight]);
                for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                    psqt[k] = vec_load_psqt(&accTilePsqtIn[k]);

                for (IndexType i = 0; i < N; ++i)
                {
                    // Difference calculation for the deactivated features
                    for (const auto index : removed[i])
                    {
                        const IndexType offset = PSQTBuckets * index + j * PsqtTileHeight;
                        auto columnPsqt = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offset]);
                        for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                            psqt[k] = vec_sub_psqt_32(psqt[k], columnPsqt[k]);
                    }

                    // Difference calculation for the activated features
                    for (const auto index : added[i])
                    {
                        const IndexType offset = PSQTBuckets * index + j * PsqtTileHeight;
                        auto columnPsqt = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offset]);
                        for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                            psqt[k] = vec_add_psqt_32(psqt[k], columnPsqt[k]);
                    }

                    // Store accumulator
                    auto accTilePsqtOut = reinterpret_cast<psqt_vec_t*>(
                      &(states_to_update[i]->*accPtr)
                         .psqtAccumulation[Perspective][j * PsqtTileHeight]);
                    for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                        vec_store_psqt(&accTilePsqtOut[k], psqt[k]);
                }
            }
        }
#else
        for (IndexType i = 0; i < N; ++i)
        {
            std::memcpy((states_to_update[i]->*accPtr).accumulation[Perspective],
                        (st->*accPtr).accumulation[Perspective], HalfDimensions * sizeof(BiasType));

            for (std::size_t k = 0; k < PSQTBuckets; ++k)
                (states_to_update[i]->*accPtr).psqtAccumulation[Perspective][k] =
                  (st->*accPtr).psqtAccumulation[Perspective][k];

            st = states_to_update[i];

            // Difference calculation for the deactivated features
            for (const auto index : removed[i])
            {
                const IndexType offset = HalfDimensions * index;
                for (IndexType j = 0; j < HalfDimensions; ++j)
                    (st->*accPtr).accumulation[Perspective][j] -= weights[offset + j];

                for (std::size_t k = 0; k < PSQTBuckets; ++k)
                    (st->*accPtr).psqtAccumulation[Perspective][k] -=
                      psqtWeights[index * PSQTBuckets + k];
            }

            // Difference calculation for the activated features
            for (const auto index : added[i])
            {
                const IndexType offset = HalfDimensions * index;
                for (IndexType j = 0; j < HalfDimensions; ++j)
                    (st->*accPtr).accumulation[Perspective][j] += weights[offset + j];

                for (std::size_t k = 0; k < PSQTBuckets; ++k)
                    (st->*accPtr).psqtAccumulation[Perspective][k] +=
                      psqtWeights[index * PSQTBuckets + k];
            }
        }
#endif
    }

    template<Color Perspective>
    void update_accumulator_refresh_cache(const Position&                           pos,
                                          AccumulatorCaches::Cache<HalfDimensions>* cache) const {
        assert(cache != nullptr);

        Square                ksq   = pos.square<KING>(Perspective);
        auto&                 entry = (*cache)[ksq][Perspective];
        FeatureSet::IndexList removed, added;

        for (Color c : {WHITE, BLACK})
        {
            for (PieceType pt = PAWN; pt <= KING; ++pt)
            {
                const Piece    piece    = make_piece(c, pt);
                const Bitboard oldBB    = entry.byColorBB[c] & entry.byTypeBB[pt];
                const Bitboard newBB    = pos.pieces(c, pt);
                Bitboard       toRemove = oldBB & ~newBB;
                Bitboard       toAdd    = newBB & ~oldBB;

                while (toRemove)
                {
                    Square sq = pop_lsb(toRemove);
                    removed.push_back(FeatureSet::make_index<Perspective>(sq, piece, ksq));
                }
                while (toAdd)
                {
                    Square sq = pop_lsb(toAdd);
                    added.push_back(FeatureSet::make_index<Perspective>(sq, piece, ksq));
                }
            }
        }

        auto& accumulator                 = pos.state()->*accPtr;
        accumulator.computed[Perspective] = true;

#ifdef VECTOR
        vec_t      acc[NumRegs];
        psqt_vec_t psqt[NumPsqtRegs];

        for (IndexType j = 0; j < HalfDimensions / TileHeight; ++j)
        {
            auto accTile =
              reinterpret_cast<vec_t*>(&accumulator.accumulation[Perspective][j * TileHeight]);
            auto entryTile = reinterpret_cast<vec_t*>(&entry.accumulation[j * TileHeight]);

            for (IndexType k = 0; k < NumRegs; ++k)
                acc[k] = entryTile[k];

            int i = 0;
            for (; i < int(std::min(removed.size(), added.size())); ++i)
            {
                IndexType       indexR  = removed[i];
                const IndexType offsetR = HalfDimensions * indexR + j * TileHeight;
                auto            columnR = reinterpret_cast<const vec_t*>(&weights[offsetR]);
                IndexType       indexA  = added[i];
                const IndexType offsetA = HalfDimensions * indexA + j * TileHeight;
                auto            columnA = reinterpret_cast<const vec_t*>(&weights[offsetA]);

                for (unsigned k = 0; k < NumRegs; ++k)
                    acc[k] = vec_add_16(acc[k], vec_sub_16(columnA[k], columnR[k]));
            }
            for (; i < int(removed.size()); ++i)
            {
                IndexType       index  = removed[i];
                const IndexType offset = HalfDimensions * index + j * TileHeight;
                auto            column = reinterpret_cast<const vec_t*>(&weights[offset]);

                for (unsigned k = 0; k < NumRegs; ++k)
                    acc[k] = vec_sub_16(acc[k], column[k]);
            }
            for (; i < int(added.size()); ++i)
            {
                IndexType       index  = added[i];
                const IndexType offset = HalfDimensions * index + j * TileHeight;
                auto            column = reinterpret_cast<const vec_t*>(&weights[offset]);

                for (unsigned k = 0; k < NumRegs; ++k)
                    acc[k] = vec_add_16(acc[k], column[k]);
            }

            for (IndexType k = 0; k < NumRegs; k++)
                vec_store(&entryTile[k], acc[k]);
            for (IndexType k = 0; k < NumRegs; k++)
                vec_store(&accTile[k], acc[k]);
        }

        for (IndexType j = 0; j < PSQTBuckets / PsqtTileHeight; ++j)
        {
            auto accTilePsqt = reinterpret_cast<psqt_vec_t*>(
              &accumulator.psqtAccumulation[Perspective][j * PsqtTileHeight]);
            auto entryTilePsqt =
              reinterpret_cast<psqt_vec_t*>(&entry.psqtAccumulation[j * PsqtTileHeight]);

            for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                psqt[k] = entryTilePsqt[k];

            for (int i = 0; i < int(removed.size()); ++i)
            {
                IndexType       index  = removed[i];
                const IndexType offset = PSQTBuckets * index + j * PsqtTileHeight;
                auto columnPsqt        = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offset]);

                for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                    psqt[k] = vec_sub_psqt_32(psqt[k], columnPsqt[k]);
            }
            for (int i = 0; i < int(added.size()); ++i)
            {
                IndexType       index  = added[i];
                const IndexType offset = PSQTBuckets * index + j * PsqtTileHeight;
                auto columnPsqt        = reinterpret_cast<const psqt_vec_t*>(&psqtWeights[offset]);

                for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                    psqt[k] = vec_add_psqt_32(psqt[k], columnPsqt[k]);
            }

            for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                vec_store_psqt(&entryTilePsqt[k], psqt[k]);
            for (std::size_t k = 0; k < NumPsqtRegs; ++k)
                vec_store_psqt(&accTilePsqt[k], psqt[k]);
        }

#else

        for (const auto index : removed)
        {
            const IndexType offset = HalfDimensions * index;
            for (IndexType j = 0; j < HalfDimensions; ++j)
                entry.accumulation[j] -= weights[offset + j];

            for (std::size_t k = 0; k < PSQTBuckets; ++k)
                entry.psqtAccumulation[k] -= psqtWeights[index * PSQTBuckets + k];
        }
        for (const auto index : added)
        {
            const IndexType offset = HalfDimensions * index;
            for (IndexType j = 0; j < HalfDimensions; ++j)
                entry.accumulation[j] += weights[offset + j];

            for (std::size_t k = 0; k < PSQTBuckets; ++k)
                entry.psqtAccumulation[k] += psqtWeights[index * PSQTBuckets + k];
        }

        // The accumulator of the refresh entry has been updated.
        // Now copy its content to the actual accumulator we were refreshing.

        std::memcpy(accumulator.accumulation[Perspective], entry.accumulation,
                    sizeof(BiasType) * HalfDimensions);

        std::memcpy(accumulator.psqtAccumulation[Perspective], entry.psqtAccumulation,
                    sizeof(int32_t) * PSQTBuckets);
#endif

        for (Color c : {WHITE, BLACK})
            entry.byColorBB[c] = pos.pieces(c);

        for (PieceType pt = PAWN; pt <= KING; ++pt)
            entry.byTypeBB[pt] = pos.pieces(pt);
    }

    template<Color Perspective>
    void hint_common_access_for_perspective(const Position&                           pos,
                                            AccumulatorCaches::Cache<HalfDimensions>* cache) const {

        // Works like update_accumulator, but performs less work.
        // Updates ONLY the accumulator for pos.

        // Look for a usable accumulator of an earlier position. We keep track
        // of the estimated gain in terms of features to be added/subtracted.
        // Fast early exit.
        if ((pos.state()->*accPtr).computed[Perspective])
            return;

        auto [oldest_st, _] = try_find_computed_accumulator<Perspective>(pos);

        if ((oldest_st->*accPtr).computed[Perspective])
        {
            // Only update current position accumulator to minimize work
            StateInfo* states_to_update[1] = {pos.state()};
            update_accumulator_incremental<Perspective, 1>(pos, oldest_st, states_to_update);
        }
        else
            update_accumulator_refresh_cache<Perspective>(pos, cache);
    }

    template<Color Perspective>
    void update_accumulator(const Position&                           pos,
                            AccumulatorCaches::Cache<HalfDimensions>* cache) const {

        auto [oldest_st, next] = try_find_computed_accumulator<Perspective>(pos);

        if ((oldest_st->*accPtr).computed[Perspective])
        {
            if (next == nullptr)
                return;

            // Now update the accumulators listed in states_to_update[], where
            // the last element is a sentinel. Currently we update two accumulators:
            //     1. for the current position
            //     2. the next accumulator after the computed one
            // The heuristic may change in the future.
            if (next == pos.state())
            {
                StateInfo* states_to_update[1] = {next};

                update_accumulator_incremental<Perspective, 1>(pos, oldest_st, states_to_update);
            }
            else
            {
                StateInfo* states_to_update[2] = {next, pos.state()};

                update_accumulator_incremental<Perspective, 2>(pos, oldest_st, states_to_update);
            }
        }
        else
            update_accumulator_refresh_cache<Perspective>(pos, cache);
    }

    template<IndexType Size>
    friend struct AccumulatorCaches::Cache;

    alignas(CacheLineSize) BiasType biases[HalfDimensions];
    alignas(CacheLineSize) WeightType weights[HalfDimensions * InputDimensions];
    alignas(CacheLineSize) PSQTWeightType psqtWeights[InputDimensions * PSQTBuckets];
};

}  // namespace Stockfish::Eval::NNUE

#endif  // #ifndef NNUE_FEATURE_TRANSFORMER_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef UCIOPTION_H_INCLUDED
#define UCIOPTION_H_INCLUDED

namespace Stockfish {
// Define a custom comparator, because the UCI options should be case-insensitive
struct CaseInsensitiveLess {
    bool operator()(const std::string&, const std::string&) const;
};

class OptionsMap;

// The Option class implements each option as specified by the UCI protocol
class Option {
   public:
    using OnChange = std::function<std::optional<std::string>(const Option&)>;

    Option(const OptionsMap*);
    Option(OnChange = nullptr);
    Option(bool v, OnChange = nullptr);
    Option(const char* v, OnChange = nullptr);
    Option(double v, int minv, int maxv, OnChange = nullptr);
    Option(const char* v, const char* cur, OnChange = nullptr);

    Option& operator=(const std::string&);
    operator int() const;
    operator std::string() const;
    bool operator==(const char*) const;
    bool operator!=(const char*) const;

    friend std::ostream& operator<<(std::ostream&, const OptionsMap&);

   private:
    friend class OptionsMap;
    friend class Engine;
    friend class Tune;

    void operator<<(const Option&);

    std::string       defaultValue, currentValue, type;
    int               min, max;
    size_t            idx;
    OnChange          on_change;
    const OptionsMap* parent = nullptr;
};

class OptionsMap {
   public:
    using InfoListener = std::function<void(std::optional<std::string>)>;

    OptionsMap()                             = default;
    OptionsMap(const OptionsMap&)            = delete;
    OptionsMap(OptionsMap&&)                 = delete;
    OptionsMap& operator=(const OptionsMap&) = delete;
    OptionsMap& operator=(OptionsMap&&)      = delete;

    void add_info_listener(InfoListener&&);

    void setoption(std::istringstream&);

    Option  operator[](const std::string&) const;
    Option& operator[](const std::string&);

    std::size_t count(const std::string&) const;

   private:
    friend class Engine;
    friend class Option;

    friend std::ostream& operator<<(std::ostream&, const OptionsMap&);

    // The options container is defined as a std::map
    using OptionsStore = std::map<std::string, Option, CaseInsensitiveLess>;

    OptionsStore options_map;
    InfoListener info;
};

}
#endif  // #ifndef UCIOPTION_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef MOVEGEN_H_INCLUDED
#define MOVEGEN_H_INCLUDED

namespace Stockfish {

class Position;

enum GenType {
    CAPTURES,
    QUIETS,
    EVASIONS,
    NON_EVASIONS,
    LEGAL
};

struct ExtMove: public Move {
    int value;

    void operator=(Move m) { data = m.raw(); }

    // Inhibit unwanted implicit conversions to Move
    // with an ambiguity that yields to a compile error.
    operator float() const = delete;
};

inline bool operator<(const ExtMove& f, const ExtMove& s) { return f.value < s.value; }

template<GenType>
ExtMove* generate(const Position& pos, ExtMove* moveList);

// The MoveList struct wraps the generate() function and returns a convenient
// list of moves. Using MoveList is sometimes preferable to directly calling
// the lower level generate() function.
template<GenType T>
struct MoveList {

    explicit MoveList(const Position& pos) :
        last(generate<T>(pos, moveList)) {}
    const ExtMove* begin() const { return moveList; }
    const ExtMove* end() const { return last; }
    size_t         size() const { return last - moveList; }
    bool           contains(Move move) const { return std::find(begin(), end(), move) != end(); }

   private:
    ExtMove moveList[MAX_MOVES], *last;
};

}  // namespace Stockfish

#endif  // #ifndef MOVEGEN_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef MOVEPICK_H_INCLUDED
#define MOVEPICK_H_INCLUDED

namespace Stockfish {

constexpr int PAWN_HISTORY_SIZE        = 512;    // has to be a power of 2
constexpr int CORRECTION_HISTORY_SIZE  = 16384;  // has to be a power of 2
constexpr int CORRECTION_HISTORY_LIMIT = 1024;

static_assert((PAWN_HISTORY_SIZE & (PAWN_HISTORY_SIZE - 1)) == 0,
              "PAWN_HISTORY_SIZE has to be a power of 2");

static_assert((CORRECTION_HISTORY_SIZE & (CORRECTION_HISTORY_SIZE - 1)) == 0,
              "CORRECTION_HISTORY_SIZE has to be a power of 2");

enum PawnHistoryType {
    Normal,
    Correction
};

template<PawnHistoryType T = Normal>
inline int pawn_structure_index(const Position& pos) {
    return pos.pawn_key() & ((T == Normal ? PAWN_HISTORY_SIZE : CORRECTION_HISTORY_SIZE) - 1);
}

// StatsEntry stores the stat table value. It is usually a number but could
// be a move or even a nested history. We use a class instead of a naked value
// to directly call history update operator<<() on the entry so to use stats
// tables at caller sites as simple multi-dim arrays.
template<typename T, int D>
class StatsEntry {

    T entry;

   public:
    void operator=(const T& v) { entry = v; }
    T*   operator&() { return &entry; }
    T*   operator->() { return &entry; }
    operator const T&() const { return entry; }

    void operator<<(int bonus) {
        static_assert(D <= std::numeric_limits<T>::max(), "D overflows T");

        // Make sure that bonus is in range [-D, D]
        int clampedBonus = std::clamp(bonus, -D, D);
        entry += clampedBonus - entry * std::abs(clampedBonus) / D;

        assert(std::abs(entry) <= D);
    }
};

// Stats is a generic N-dimensional array used to store various statistics.
// The first template parameter T is the base type of the array, and the second
// template parameter D limits the range of updates in [-D, D] when we update
// values with the << operator, while the last parameters (Size and Sizes)
// encode the dimensions of the array.
template<typename T, int D, int Size, int... Sizes>
struct Stats: public std::array<Stats<T, D, Sizes...>, Size> {
    using stats = Stats<T, D, Size, Sizes...>;

    void fill(const T& v) {

        // For standard-layout 'this' points to the first struct member
        assert(std::is_standard_layout_v<stats>);

        using entry = StatsEntry<T, D>;
        entry* p    = reinterpret_cast<entry*>(this);
        std::fill(p, p + sizeof(*this) / sizeof(entry), v);
    }
};

template<typename T, int D, int Size>
struct Stats<T, D, Size>: public std::array<StatsEntry<T, D>, Size> {};

// In stats table, D=0 means that the template parameter is not used
enum StatsParams {
    NOT_USED = 0
};
enum StatsType {
    NoCaptures,
    Captures
};

// ButterflyHistory records how often quiet moves have been successful or unsuccessful
// during the current search, and is used for reduction and move ordering decisions.
// It uses 2 tables (one for each color) indexed by the move's from and to squares,
// see www.chessprogramming.org/Butterfly_Boards (~11 elo)
using ButterflyHistory = Stats<int16_t, 7183, COLOR_NB, int(SQUARE_NB) * int(SQUARE_NB)>;

// CapturePieceToHistory is addressed by a move's [piece][to][captured piece type]
using CapturePieceToHistory = Stats<int16_t, 10692, PIECE_NB, SQUARE_NB, PIECE_TYPE_NB>;

// PieceToHistory is like ButterflyHistory but is addressed by a move's [piece][to]
using PieceToHistory = Stats<int16_t, 29952, PIECE_NB, SQUARE_NB>;

// ContinuationHistory is the combined history of a given pair of moves, usually
// the current one given a previous one. The nested history table is based on
// PieceToHistory instead of ButterflyBoards.
// (~63 elo)
using ContinuationHistory = Stats<PieceToHistory, NOT_USED, PIECE_NB, SQUARE_NB>;

// PawnHistory is addressed by the pawn structure and a move's [piece][to]
using PawnHistory = Stats<int16_t, 8192, PAWN_HISTORY_SIZE, PIECE_NB, SQUARE_NB>;

// CorrectionHistory is addressed by color and pawn structure
using CorrectionHistory =
  Stats<int16_t, CORRECTION_HISTORY_LIMIT, COLOR_NB, CORRECTION_HISTORY_SIZE>;

// The MovePicker class is used to pick one pseudo-legal move at a time from the
// current position. The most important method is next_move(), which emits one
// new pseudo-legal move on every call, until there are no moves left, when
// Move::none() is returned. In order to improve the efficiency of the alpha-beta
// algorithm, MovePicker attempts to return the moves which are most likely to get
// a cut-off first.
class MovePicker {

    enum PickType {
        Next,
        Best
    };

   public:
    MovePicker(const MovePicker&)            = delete;
    MovePicker& operator=(const MovePicker&) = delete;
    MovePicker(const Position&,
               Move,
               Depth,
               const ButterflyHistory*,
               const CapturePieceToHistory*,
               const PieceToHistory**,
               const PawnHistory*);
    MovePicker(const Position&, Move, int, const CapturePieceToHistory*);
    Move next_move(bool skipQuiets = false);

   private:
    template<PickType T, typename Pred>
    Move select(Pred);
    template<GenType>
    void     score();
    ExtMove* begin() { return cur; }
    ExtMove* end() { return endMoves; }

    const Position&              pos;
    const ButterflyHistory*      mainHistory;
    const CapturePieceToHistory* captureHistory;
    const PieceToHistory**       continuationHistory;
    const PawnHistory*           pawnHistory;
    Move                         ttMove;
    ExtMove *                    cur, *endMoves, *endBadCaptures, *beginBadQuiets, *endBadQuiets;
    int                          stage;
    int                          threshold;
    Depth                        depth;
    ExtMove                      moves[MAX_MOVES];
};

}  // namespace Stockfish

#endif  // #ifndef MOVEPICK_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef TBPROBE_H
#define TBPROBE_H

namespace Stockfish {
class Position;
class OptionsMap;

using Depth = int;

namespace Search {
struct RootMove;
using RootMoves = std::vector<RootMove>;
}
}

namespace Stockfish::Tablebases {

struct Config {
    int   cardinality = 0;
    bool  rootInTB    = false;
    bool  useRule50   = false;
    Depth probeDepth  = 0;
};

enum WDLScore {
    WDLLoss        = -2,  // Loss
    WDLBlessedLoss = -1,  // Loss, but draw under 50-move rule
    WDLDraw        = 0,   // Draw
    WDLCursedWin   = 1,   // Win, but draw under 50-move rule
    WDLWin         = 2,   // Win
};

// Possible states after a probing operation
enum ProbeState {
    FAIL              = 0,   // Probe failed (missing file table)
    OK                = 1,   // Probe successful
    CHANGE_STM        = -1,  // DTZ should check the other side
    ZEROING_BEST_MOVE = 2    // Best move zeroes DTZ (capture or pawn move)
};

extern int MaxCardinality;

void     init(const std::string& paths);
WDLScore probe_wdl(Position& pos, ProbeState* result);
int      probe_dtz(Position& pos, ProbeState* result);
bool     root_probe(Position& pos, Search::RootMoves& rootMoves, bool rule50, bool rankDTZ);
bool     root_probe_wdl(Position& pos, Search::RootMoves& rootMoves, bool rule50);
Config   rank_root_moves(const OptionsMap&  options,
                         Position&          pos,
                         Search::RootMoves& rootMoves,
                         bool               rankDTZ = false);

}  // namespace Stockfish::Tablebases

#endif

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef SEARCH_H_INCLUDED
#define SEARCH_H_INCLUDED

namespace Stockfish {

// Different node types, used as a template parameter
enum NodeType {
    NonPV,
    PV,
    Root
};

class TranspositionTable;
class ThreadPool;
class OptionsMap;

namespace Search {

// Stack struct keeps track of the information we need to remember from nodes
// shallower and deeper in the tree during the search. Each search thread has
// its own array of Stack objects, indexed by the current ply.
struct Stack {
    Move*           pv;
    PieceToHistory* continuationHistory;
    int             ply;
    Move            currentMove;
    Move            excludedMove;
    Value           staticEval;
    int             statScore;
    int             moveCount;
    bool            inCheck;
    bool            ttPv;
    bool            ttHit;
    int             cutoffCnt;
};

// RootMove struct is used for moves at the root of the tree. For each root move
// we store a score and a PV (really a refutation in the case of moves which
// fail low). Score is normally set at -VALUE_INFINITE for all non-pv moves.
struct RootMove {

    explicit RootMove(Move m) :
        pv(1, m) {}
    bool extract_ponder_from_tt(const TranspositionTable& tt, Position& pos);
    bool operator==(const Move& m) const { return pv[0] == m; }
    // Sort in descending order
    bool operator<(const RootMove& m) const {
        return m.score != score ? m.score < score : m.previousScore < previousScore;
```

```cpp
    }

    uint64_t          effort          = 0;
    Value             score           = -VALUE_INFINITE;
    Value             previousScore   = -VALUE_INFINITE;
    Value             averageScore    = -VALUE_INFINITE;
    Value             uciScore        = -VALUE_INFINITE;
    bool              scoreLowerbound = false;
    bool              scoreUpperbound = false;
    int               selDepth        = 0;
    int               tbRank          = 0;
    Value             tbScore;
    std::vector<Move> pv;
};

using RootMoves = std::vector<RootMove>;

// LimitsType struct stores information sent by the caller about the analysis required.
struct LimitsType {

    // Init explicitly due to broken value-initialization of non POD in MSVC
    LimitsType() {
        time[WHITE] = time[BLACK] = inc[WHITE] = inc[BLACK] = npmsec = movetime = TimePoint(0);
        movestogo = depth = mate = perft = infinite = 0;
        nodes                                       = 0;
        ponderMode                                  = false;
    }

    bool use_time_management() const { return time[WHITE] || time[BLACK]; }

    std::vector<std::string> searchmoves;
    TimePoint                time[COLOR_NB], inc[COLOR_NB], npmsec, movetime, startTime;
    int                      movestogo, depth, mate, perft, infinite;
    uint64_t                 nodes;
    bool                     ponderMode;
    Square                   capSq;
};

// The UCI stores the uci options, thread pool, and transposition table.
// This struct is used to easily forward data to the Search::Worker class.
struct SharedState {
    SharedState(const OptionsMap&                               optionsMap,
                ThreadPool&                                     threadPool,
                TranspositionTable&                             transpositionTable,
                const LazyNumaReplicated<Eval::NNUE::Networks>& nets) :
        options(optionsMap),
        threads(threadPool),
        tt(transpositionTable),
        networks(nets) {}

    const OptionsMap&                               options;
    ThreadPool&                                     threads;
    TranspositionTable&                             tt;
    const LazyNumaReplicated<Eval::NNUE::Networks>& networks;
};

class Worker;

// Null Object Pattern, implement a common interface for the SearchManagers.
// A Null Object will be given to non-mainthread workers.
class ISearchManager {
   public:
    virtual ~ISearchManager() {}
    virtual void check_time(Search::Worker&) = 0;
};

struct InfoShort {
    int   depth;
    Score score;
};

struct InfoFull: InfoShort {
    int              selDepth;
    size_t           multiPV;
    std::string_view wdl;
    std::string_view bound;
    size_t           timeMs;
    size_t           nodes;
    size_t           nps;
    size_t           tbHits;
    std::string_view pv;
    int              hashfull;
};

struct InfoIteration {
    int              depth;
    std::string_view currmove;
    size_t           currmovenumber;
};

// Skill structure is used to implement strength limit. If we have a UCI_Elo,
// we convert it to an appropriate skill level, anchored to the Stash engine.
// This method is based on a fit of the Elo results for games played between
// Stockfish at various skill levels and various versions of the Stash engine.
// Skill 0 .. 19 now covers CCRL Blitz Elo from 1320 to 3190, approximately
// Reference: https://github.com/vondele/Stockfish/commit/a08b8d4e9711c2
struct Skill {
    // Lowest and highest Elo ratings used in the skill level calculation
    constexpr static int LowestElo  = 1320;
    constexpr static int HighestElo = 3190;

    Skill(int skill_level, int uci_elo) {
        if (uci_elo)
        {
            double e = double(uci_elo - LowestElo) / (HighestElo - LowestElo);
            level = std::clamp((((37.2473 * e - 40.8525) * e + 22.2943) * e - 0.311438), 0.0, 19.0);
        }
        else
            level = double(skill_level);
    }
    bool enabled() const { return level < 20.0; }
    bool time_to_pick(Depth depth) const { return depth == 1 + int(level); }
    Move pick_best(const RootMoves&, size_t multiPV);

    double level;
    Move   best = Move::none();
};

// SearchManager manages the search from the main thread. It is responsible for
// keeping track of the time, and storing data strictly related to the main thread.
class SearchManager: public ISearchManager {
   public:
    using UpdateShort    = std::function<void(const InfoShort&)>;
    using UpdateFull     = std::function<void(const InfoFull&)>;
    using UpdateIter     = std::function<void(const InfoIteration&)>;
    using UpdateBestmove = std::function<void(std::string_view, std::string_view)>;

    struct UpdateContext {
        UpdateShort    onUpdateNoMoves;
        UpdateFull     onUpdateFull;
        UpdateIter     onIter;
        UpdateBestmove onBestmove;
    };

    SearchManager(const UpdateContext& updateContext) :
        updates(updateContext) {}

    void check_time(Search::Worker& worker) override;

    void pv(Search::Worker&           worker,
            const ThreadPool&         threads,
            const TranspositionTable& tt,
            Depth                     depth);

    Stockfish::TimeManagement tm;
    double                    originalTimeAdjust;
    int                       callsCnt;
    std::atomic_bool          ponder;

    std::array<Value, 4> iterValue;
    double               previousTimeReduction;
    Value                bestPreviousScore;
    Value                bestPreviousAverageScore;
    bool                 stopOnPonderhit;

    size_t id;

    const UpdateContext& updates;
};

class NullSearchManager: public ISearchManager {
   public:
    void check_time(Search::Worker&) override {}
};

// Search::Worker is the class that does the actual search.
// It is instantiated once per thread, and it is responsible for keeping track
// of the search history, and storing data required for the search.
class Worker {
   public:
    Worker(SharedState&, std::unique_ptr<ISearchManager>, size_t, NumaReplicatedAccessToken);

    // Called at instantiation to initialize reductions tables.
    // Reset histories, usually before a new game.
    void clear();

    // Called when the program receives the UCI 'go' command.
    // It searches from the root position and outputs the "bestmove".
    void start_searching();

    bool is_mainthread() const { return threadIdx == 0; }

    void ensure_network_replicated();

    // Public because they need to be updatable by the stats
    ButterflyHistory      mainHistory;
    CapturePieceToHistory captureHistory;
    ContinuationHistory   continuationHistory[2][2];
    PawnHistory           pawnHistory;
    CorrectionHistory     correctionHistory;

   private:
    void iterative_deepening();

    // This is the main search function, for both PV and non-PV nodes
    template<NodeType nodeType>
    Value search(Position& pos, Stack* ss, Value alpha, Value beta, Depth depth, bool cutNode);

    // Quiescence search function, which is called by the main search
    template<NodeType nodeType>
    Value qsearch(Position& pos, Stack* ss, Value alpha, Value beta);

    Depth reduction(bool i, Depth d, int mn, int delta) const;

    // Pointer to the search manager, only allowed to be called by the main thread
    SearchManager* main_manager() const {
        assert(threadIdx == 0);
        return static_cast<SearchManager*>(manager.get());
    }

    TimePoint elapsed() const;
    TimePoint elapsed_time() const;

    LimitsType limits;

    size_t                pvIdx, pvLast;
    std::atomic<uint64_t> nodes, tbHits, bestMoveChanges;
    int                   selDepth, nmpMinPly;

    Value optimism[COLOR_NB];

    Position  rootPos;
    StateInfo rootState;
    RootMoves rootMoves;
    Depth     rootDepth, completedDepth;
    Value     rootDelta;

    size_t                    threadIdx;
    NumaReplicatedAccessToken numaAccessToken;

    // Reductions lookup table initialized at startup
    std::array<int, MAX_MOVES> reductions;  // [depth or moveNumber]

    // The main thread has a SearchManager, the others have a NullSearchManager
    std::unique_ptr<ISearchManager> manager;

    Tablebases::Config tbConfig;

    const OptionsMap&                               options;
    ThreadPool&                                     threads;
    TranspositionTable&                             tt;
    const LazyNumaReplicated<Eval::NNUE::Networks>& networks;

    // Used by NNUE
    Eval::NNUE::AccumulatorCaches refreshTable;

    friend class Stockfish::ThreadPool;
    friend class SearchManager;
};

}  // namespace Search

}  // namespace Stockfish

#endif  // #ifndef SEARCH_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef THREAD_WIN32_OSX_H_INCLUDED
#define THREAD_WIN32_OSX_H_INCLUDED

// On OSX threads other than the main thread are created with a reduced stack
// size of 512KB by default, this is too low for deep searches, which require
// somewhat more than 1MB stack, so adjust it to TH_STACK_SIZE.
// The implementation calls pthread_create() with the stack size parameter
// equal to the Linux 8MB default, on platforms that support it.

#if defined(__APPLE__) || defined(__MINGW32__) || defined(__MINGW64__) || defined(USE_PTHREADS)

namespace Stockfish {

class NativeThread {
    pthread_t thread;

    static constexpr size_t TH_STACK_SIZE = 8 * 1024 * 1024;

   public:
    template<class Function, class... Args>
    explicit NativeThread(Function&& fun, Args&&... args) {
        auto func = new std::function<void()>(
          std::bind(std::forward<Function>(fun), std::forward<Args>(args)...));

        pthread_attr_t attr_storage, *attr = &attr_storage;
        pthread_attr_init(attr);
        pthread_attr_setstacksize(attr, TH_STACK_SIZE);

        auto start_routine = [](void* ptr) -> void* {
            auto f = reinterpret_cast<std::function<void()>*>(ptr);
            // Call the function
            (*f)();
            delete f;
            return nullptr;
        };

        pthread_create(&thread, attr, start_routine, func);
    }

    void join() { pthread_join(thread, nullptr); }
};

}  // namespace Stockfish

#else  // Default case: use STL classes

namespace Stockfish {

using NativeThread = std::thread;

}  // namespace Stockfish

#endif

#endif  // #ifndef THREAD_WIN32_OSX_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef THREAD_H_INCLUDED
#define THREAD_H_INCLUDED

namespace Stockfish {

class OptionsMap;
using Value = int;

// Sometimes we don't want to actually bind the threads, but the recipient still
// needs to think it runs on *some* NUMA node, such that it can access structures
// that rely on NUMA node knowledge. This class encapsulates this optional process
// such that the recipient does not need to know whether the binding happened or not.
class OptionalThreadToNumaNodeBinder {
   public:
    OptionalThreadToNumaNodeBinder(NumaIndex n) :
        numaConfig(nullptr),
        numaId(n) {}

    OptionalThreadToNumaNodeBinder(const NumaConfig& cfg, NumaIndex n) :
        numaConfig(&cfg),
        numaId(n) {}

    NumaReplicatedAccessToken operator()() const {
        if (numaConfig != nullptr)
            return numaConfig->bind_current_thread_to_numa_node(numaId);
        else
            return NumaReplicatedAccessToken(numaId);
    }

   private:
    const NumaConfig* numaConfig;
    NumaIndex         numaId;
};

// Abstraction of a thread. It contains a pointer to the worker and a native thread.
// After construction, the native thread is started with idle_loop()
// waiting for a signal to start searching.
// When the signal is received, the thread starts searching and when
// the search is finished, it goes back to idle_loop() waiting for a new signal.
class Thread {
   public:
    Thread(Search::SharedState&,
           std::unique_ptr<Search::ISearchManager>,
           size_t,
           OptionalThreadToNumaNodeBinder);
    virtual ~Thread();

    void idle_loop();
    void start_searching();
    void clear_worker();
    void run_custom_job(std::function<void()> f);

    void ensure_network_replicated();

    // Thread has been slightly altered to allow running custom jobs, so
    // this name is no longer correct. However, this class (and ThreadPool)
    // require further work to make them properly generic while maintaining
    // appropriate specificity regarding search, from the point of view of an
    // outside user, so renaming of this function is left for whenever that happens.
    void   wait_for_search_finished();
    size_t id() const { return idx; }

    std::unique_ptr<Search::Worker> worker;
    std::function<void()>           jobFunc;

   private:
    std::mutex                mutex;
    std::condition_variable   cv;
    size_t                    idx, nthreads;
    bool                      exit = false, searching = true;  // Set before starting std::thread
    NativeThread              stdThread;
    NumaReplicatedAccessToken numaAccessToken;
};

// ThreadPool struct handles all the threads-related stuff like init, starting,
// parking and, most importantly, launching a thread. All the access to threads
// is done through this class.
class ThreadPool {
   public:
    ThreadPool() {}

    ~ThreadPool() {
        // destroy any existing thread(s)
        if (threads.size() > 0)
        {
            main_thread()->wait_for_search_finished();

            threads.clear();
        }
    }

    ThreadPool(const ThreadPool&) = delete;
    ThreadPool(ThreadPool&&)      = delete;

    ThreadPool& operator=(const ThreadPool&) = delete;
    ThreadPool& operator=(ThreadPool&&)      = delete;

    void   start_thinking(const OptionsMap&, Position&, StateListPtr&, Search::LimitsType);
    void   run_on_thread(size_t threadId, std::function<void()> f);
    void   wait_on_thread(size_t threadId);
    size_t num_threads() const;
    void   clear();
    void   set(const NumaConfig& numaConfig,
               Search::SharedState,
               const Search::SearchManager::UpdateContext&);

    Search::SearchManager* main_manager();
    Thread*                main_thread() const { return threads.front().get(); }
    uint64_t               nodes_searched() const;
    uint64_t               tb_hits() const;
    Thread*                get_best_thread() const;
    void                   start_searching();
    void                   wait_for_search_finished() const;

    std::vector<size_t> get_bound_thread_count_by_numa_node() const;

    void ensure_network_replicated();

    std::atomic_bool stop, abortedSearch, increaseDepth;

    auto cbegin() const noexcept { return threads.cbegin(); }
    auto begin() noexcept { return threads.begin(); }
    auto end() noexcept { return threads.end(); }
    auto cend() const noexcept { return threads.cend(); }
    auto size() const noexcept { return threads.size(); }
    auto empty() const noexcept { return threads.empty(); }

   private:
    StateListPtr                         setupStates;
    std::vector<std::unique_ptr<Thread>> threads;
    std::vector<NumaIndex>               boundThreadToNumaNode;

    uint64_t accumulate(std::atomic<uint64_t> Search::Worker::*member) const {

        uint64_t sum = 0;
        for (auto&& th : threads)
            sum += (th->worker.get()->*member).load(std::memory_order_relaxed);
        return sum;
    }
};

}  // namespace Stockfish

#endif  // #ifndef THREAD_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef TT_H_INCLUDED
#define TT_H_INCLUDED

namespace Stockfish {

class ThreadPool;
struct TTEntry;
struct Cluster;

// There is only one global hash table for the engine and all its threads. For chess in particular, we even allow racy
// updates between threads to and from the TT, as taking the time to synchronize access would cost thinking time and
// thus elo. As a hash table, collisions are possible and may cause chess playing issues (bizarre blunders, faulty mate
// reports, etc). Fixing these also loses elo; however such risk decreases quickly with larger TT size.
//
// `probe` is the primary method: given a board position, we lookup its entry in the table, and return a tuple of:
//   1) whether the entry already has this position
//   2) a copy of the prior data (if any) (may be inconsistent due to read races)
//   3) a writer object to this entry
// The copied data and the writer are separated to maintain clear boundaries between local vs global objects.

// A copy of the data already in the entry (possibly collided). `probe` may be racy, resulting in inconsistent data.
struct TTData {
    Move  move;
    Value value, eval;
    Depth depth;
    Bound bound;
    bool  is_pv;
};

// This is used to make racy writes to the global TT.
struct TTWriter {
   public:
    void write(Key k, Value v, bool pv, Bound b, Depth d, Move m, Value ev, uint8_t generation8);

   private:
    friend class TranspositionTable;
    TTEntry* entry;
    TTWriter(TTEntry* tte);
};

class TranspositionTable {

   public:
    ~TranspositionTable() { aligned_large_pages_free(table); }

    void resize(size_t mbSize, ThreadPool& threads);  // Set TT size
    void clear(ThreadPool& threads);                  // Re-initialize memory, multithreaded
    int  hashfull()
      const;  // Approximate what fraction of entries (permille) have been written to during this root search

    void
    new_search();  // This must be called at the beginning of each root search to track entry aging
    uint8_t generation() const;  // The current age, used when writing new data to the TT
    std::tuple<bool, TTData, TTWriter>
    probe(const Key key) const;  // The main method, whose retvals separate local vs global objects
    TTEntry* first_entry(const Key key)
      const;  // This is the hash function; its only external use is memory prefetching.

   private:
    friend struct TTEntry;

    size_t   clusterCount;
    Cluster* table = nullptr;

    uint8_t generation8 = 0;  // Size must be not bigger than TTEntry::genBound8
};

}  // namespace Stockfish

#endif  // #ifndef TT_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef BENCHMARK_H_INCLUDED
#define BENCHMARK_H_INCLUDED

namespace Stockfish::Benchmark {

std::vector<std::string> setup_bench(const std::string&, std::istream&);

}  // namespace Stockfish

#endif  // #ifndef BENCHMARK_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef ENGINE_H_INCLUDED
#define ENGINE_H_INCLUDED

namespace Stockfish {

enum Square : int;

class Engine {
   public:
    using InfoShort = Search::InfoShort;
    using InfoFull  = Search::InfoFull;
    using InfoIter  = Search::InfoIteration;

    Engine(std::string path = "");

    // Cannot be movable due to components holding backreferences to fields
    Engine(const Engine&)            = delete;
    Engine(Engine&&)                 = delete;
    Engine& operator=(const Engine&) = delete;
    Engine& operator=(Engine&&)      = delete;

    ~Engine() { wait_for_search_finished(); }

    std::uint64_t perft(const std::string& fen, Depth depth, bool isChess960);

    // non blocking call to start searching
    void go(Search::LimitsType&);
    // non blocking call to stop searching
    void stop();

    // blocking call to wait for search to finish
    void wait_for_search_finished();
    // set a new position, moves are in UCI format
    void set_position(const std::string& fen, const std::vector<std::string>& moves);

    // modifiers

    void set_numa_config_from_option(const std::string& o);
    void resize_threads();
    void set_tt_size(size_t mb);
    void set_ponderhit(bool);
    void search_clear();

    void set_on_update_no_moves(std::function<void(const InfoShort&)>&&);
    void set_on_update_full(std::function<void(const InfoFull&)>&&);
    void set_on_iter(std::function<void(const InfoIter&)>&&);
    void set_on_bestmove(std::function<void(std::string_view, std::string_view)>&&);

    // network related

    void verify_networks() const;
    void load_networks();
    void load_big_network(const std::string& file);
    void load_small_network(const std::string& file);
    void save_network(const std::pair<std::optional<std::string>, std::string> files[2]);

    // utility functions

    void trace_eval() const;

    const OptionsMap& get_options() const;
    OptionsMap&       get_options();

    std::string                            fen() const;
    void                                   flip();
    std::string                            visualize() const;
    std::vector<std::pair<size_t, size_t>> get_bound_thread_count_by_numa_node() const;
    std::string                            get_numa_config_as_string() const;
    std::string                            numa_config_information_as_string() const;
    std::string                            thread_binding_information_as_string() const;

   private:
    const std::string binaryDirectory;

    NumaReplicationContext numaContext;

    Position     pos;
    StateListPtr states;
    Square       capSq;

    OptionsMap                               options;
    ThreadPool                               threads;
    TranspositionTable                       tt;
    LazyNumaReplicated<Eval::NNUE::Networks> networks;

    Search::SearchManager::UpdateContext updateContext;
};

}  // namespace Stockfish

#endif  // #ifndef ENGINE_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef UCI_H_INCLUDED
#define UCI_H_INCLUDED

namespace Stockfish {

class Position;
class Move;
class Score;
enum Square : int;
using Value = int;

class UCIEngine {
   public:
    UCIEngine(int argc, char** argv);

    void loop();

    static int         to_cp(Value v, const Position& pos);
    static std::string format_score(const Score& s);
    static std::string square(Square s);
    static std::string move(Move m, bool chess960);
    static std::string wdl(Value v, const Position& pos);
    static std::string to_lower(std::string str);
    static Move        to_move(const Position& pos, std::string str);

    static Search::LimitsType parse_limits(std::istream& is);

    auto& engine_options() { return engine.get_options(); }

    Engine      engine;
    CommandLine cli;

    static void print_info_string(const std::string& str);

    void          go(std::istringstream& is);
    void          bench(std::istream& args);
    void          position(std::istringstream& is);
    void          setoption(std::istringstream& is);
    std::uint64_t perft(const Search::LimitsType&);

    static void on_update_no_moves(const Engine::InfoShort& info);
    static void on_update_full(const Engine::InfoFull& info, bool showWDL);
    static void on_iter(const Engine::InfoIter& info);
    static void on_bestmove(std::string_view bestmove, std::string_view ponder);
};

}  // namespace Stockfish

#endif  // #ifndef UCI_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef PERFT_H_INCLUDED
#define PERFT_H_INCLUDED

namespace Stockfish::Benchmark {

// Utility to verify move generation. All the leaf nodes up
// to the given depth are generated and counted, and the sum is returned.
template<bool Root>
uint64_t perft(Position& pos, Depth depth) {

    StateInfo st;
    ASSERT_ALIGNED(&st, Eval::NNUE::CacheLineSize);

    uint64_t   cnt, nodes = 0;
    const bool leaf = (depth == 2);

    for (const auto& m : MoveList<LEGAL>(pos))
    {
        if (Root && depth <= 1)
            cnt = 1, nodes++;
        else
        {
            pos.do_move(m, st);
            cnt = leaf ? MoveList<LEGAL>(pos).size() : perft<false>(pos, depth - 1);
            nodes += cnt;
            pos.undo_move(m);
        }
        if (Root)
            sync_cout << UCIEngine::move(m, pos.is_chess960()) << ": " << cnt << sync_endl;
    }
    return nodes;
}

inline uint64_t perft(const std::string& fen, Depth depth, bool isChess960) {
    StateListPtr states(new std::deque<StateInfo>(1));
    Position     p;
    p.set(fen, isChess960, &states->back());

    return perft<true>(p, depth);
}
}

#endif  // PERFT_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef NNUE_MISC_H_INCLUDED
#define NNUE_MISC_H_INCLUDED

namespace Stockfish {

class Position;

namespace Eval::NNUE {

struct EvalFile {
    // Default net name, will use one of the EvalFileDefaultName* macros defined
    // in evaluate.h
    std::string defaultName;
    // Selected net name, either via uci option or default
    std::string current;
    // Net description extracted from the net file
    std::string netDescription;
};

struct NnueEvalTrace {
    static_assert(LayerStacks == PSQTBuckets);

    Value       psqt[LayerStacks];
    Value       positional[LayerStacks];
    std::size_t correctBucket;
};

struct Networks;
struct AccumulatorCaches;

std::string trace(Position& pos, const Networks& networks, AccumulatorCaches& caches);
void        hint_common_parent_position(const Position&    pos,
                                        const Networks&    networks,
                                        AccumulatorCaches& caches);

}  // namespace Stockfish::Eval::NNUE
}  // namespace Stockfish

#endif  // #ifndef NNUE_MISC_H_INCLUDED

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef NETWORK_H_INCLUDED
#define NETWORK_H_INCLUDED
```

```
namespace Stockfish::Eval::NNUE {

enum class EmbeddedNNUEType {
    BIG,
    SMALL,
};

using NetworkOutput = std::tuple<Value, Value>;

template<typename Arch, typename Transformer>
class Network {
    static constexpr IndexType FTDimensions = Arch::TransformedFeatureDimensions;

   public:
    Network(EvalFile file, EmbeddedNNUEType type) :
        evalFile(file),
        embeddedType(type) {}

    Network(const Network& other);
    Network(Network&& other) = default;

    Network& operator=(const Network& other);
    Network& operator=(Network&& other) = default;

    void load(const std::string& rootDirectory, std::string evalfilePath);
    bool save(const std::optional<std::string>& filename) const;

    NetworkOutput evaluate(const Position&                         pos,
                           AccumulatorCaches::Cache<FTDimensions>* cache) const;

    void hint_common_access(const Position&                         pos,
                            AccumulatorCaches::Cache<FTDimensions>* cache) const;

    void          verify(std::string evalfilePath) const;
    NnueEvalTrace trace_evaluate(const Position&                         pos,
                                 AccumulatorCaches::Cache<FTDimensions>* cache) const;

   private:
    void load_user_net(const std::string&, const std::string&);
    void load_internal();

    void initialize();

    bool                       save(std::ostream&, const std::string&, const std::string&) const;
    std::optional<std::string> load(std::istream&);

    bool read_header(std::istream&, std::uint32_t*, std::string*) const;
    bool write_header(std::ostream&, std::uint32_t, const std::string&) const;

    bool read_parameters(std::istream&, std::string&) const;
    bool write_parameters(std::ostream&, const std::string&) const;

    // Input feature converter
    LargePagePtr<Transformer> featureTransformer;

    // Evaluation function
    AlignedPtr<Arch[]> network;

    EvalFile         evalFile;
    EmbeddedNNUEType embeddedType;

    // Hash value of evaluation function structure
    static constexpr std::uint32_t hash = Transformer::get_hash_value() ^ Arch::get_hash_value();

    template<IndexType Size>
    friend struct AccumulatorCaches::Cache;
};

// Definitions of the network types
using SmallFeatureTransformer =
  FeatureTransformer<TransformedFeatureDimensionsSmall, &StateInfo::accumulatorSmall>;
using SmallNetworkArchitecture =
  NetworkArchitecture<TransformedFeatureDimensionsSmall, L2Small, L3Small>;

using BigFeatureTransformer =
  FeatureTransformer<TransformedFeatureDimensionsBig, &StateInfo::accumulatorBig>;
using BigNetworkArchitecture = NetworkArchitecture<TransformedFeatureDimensionsBig, L2Big, L3Big>;

using NetworkBig   = Network<BigNetworkArchitecture, BigFeatureTransformer>;
using NetworkSmall = Network<SmallNetworkArchitecture, SmallFeatureTransformer>;

struct Networks {
    Networks(NetworkBig&& nB, NetworkSmall&& nS) :
        big(std::move(nB)),
        small(std::move(nS)) {}

    NetworkBig   big;
    NetworkSmall small;
};

}  // namespace Stockfish

#endif
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

namespace NN = Eval::NNUE;

constexpr int  MaxHashMB = Is64Bit ? 33554432 : 2048;

Engine::Engine(std::string path) :
    binaryDirectory(CommandLine::get_binary_directory(path)),
    numaContext(NumaConfig::from_system()),
    states(new std::deque<StateInfo>(1)),
    threads(),
    networks(
      numaContext,
      NN::Networks(
        NN::NetworkBig({EvalFileDefaultNameBig, "None", ""}, NN::EmbeddedNNUEType::BIG),
        NN::NetworkSmall({EvalFileDefaultNameSmall, "None", ""}, NN::EmbeddedNNUEType::SMALL))) {
    pos.set(StartFEN, false, &states->back());
    capSq = SQ_NONE;

    options["Debug Log File"] << Option("", [](const Option& o) {
        start_logger(o);
        return std::nullopt;
    });

    options["NumaPolicy"] << Option("auto", [this](const Option& o) {
        set_numa_config_from_option(o);
        return numa_config_information_as_string() + "\n" + thread_binding_information_as_string();
    });

    options["Threads"] << Option(1, 1, 1024, [this](const Option&) {
        resize_threads();
        return thread_binding_information_as_string();
    });

    options["Hash"] << Option(16, 1, MaxHashMB, [this](const Option& o) {
        set_tt_size(o);
        return std::nullopt;
    });

    options["Clear Hash"] << Option([this](const Option&) {
        search_clear();
        return std::nullopt;
    });
    options["Ponder"] << Option(false);
    options["MultiPV"] << Option(1, 1, MAX_MOVES);
    options["Skill Level"] << Option(20, 0, 20);
    options["Move Overhead"] << Option(10, 0, 5000);
    options["nodestime"] << Option(0, 0, 10000);
    options["UCI_Chess960"] << Option(false);
    options["UCI_LimitStrength"] << Option(false);
    options["UCI_Elo"] << Option(Stockfish::Search::Skill::LowestElo,
                                 Stockfish::Search::Skill::LowestElo,
                                 Stockfish::Search::Skill::HighestElo);
    options["UCI_ShowWDL"] << Option(false);
    options["SyzygyPath"] << Option("", [](const Option& o) {
        Tablebases::init(o);
        return std::nullopt;
    });
    options["SyzygyProbeDepth"] << Option(1, 1, 100);
    options["Syzygy50MoveRule"] << Option(true);
    options["SyzygyProbeLimit"] << Option(7, 0, 7);
    options["EvalFile"] << Option(EvalFileDefaultNameBig, [this](const Option& o) {
        load_big_network(o);
        return std::nullopt;
    });
    options["EvalFileSmall"] << Option(EvalFileDefaultNameSmall, [this](const Option& o) {
        load_small_network(o);
        return std::nullopt;
    });

    load_networks();
    resize_threads();
}

std::uint64_t Engine::perft(const std::string& fen, Depth depth, bool isChess960) {
    verify_networks();

    return Benchmark::perft(fen, depth, isChess960);
}

void Engine::go(Search::LimitsType& limits) {
    assert(limits.perft == 0);
    verify_networks();
    limits.capSq = capSq;

    threads.start_thinking(options, pos, states, limits);
}
void Engine::stop() { threads.stop = true; }

void Engine::search_clear() {
    wait_for_search_finished();

    tt.clear(threads);
    threads.clear();

    // @TODO wont work with multiple instances
    Tablebases::init(options["SyzygyPath"]);  // Free mapped files
}

void Engine::set_on_update_no_moves(std::function<void(const Engine::InfoShort&)>&& f) {
    updateContext.onUpdateNoMoves = std::move(f);
}

void Engine::set_on_update_full(std::function<void(const Engine::InfoFull&)>&& f) {
    updateContext.onUpdateFull = std::move(f);
}

void Engine::set_on_iter(std::function<void(const Engine::InfoIter&)>&& f) {
    updateContext.onIter = std::move(f);
}

void Engine::set_on_bestmove(std::function<void(std::string_view, std::string_view)>&& f) {
    updateContext.onBestmove = std::move(f);
}

void Engine::wait_for_search_finished() { threads.main_thread()->wait_for_search_finished(); }

void Engine::set_position(const std::string& fen, const std::vector<std::string>& moves) {
    // Drop the old state and create a new one
    states = StateListPtr(new std::deque<StateInfo>(1));
    pos.set(fen, options["UCI_Chess960"], &states->back());

    capSq = SQ_NONE;
    for (const auto& move : moves)
    {
        auto m = UCIEngine::to_move(pos, move);

        if (m == Move::none())
            break;

        states->emplace_back();
        pos.do_move(m, states->back());

        capSq          = SQ_NONE;
        DirtyPiece& dp = states->back().dirtyPiece;
        if (dp.dirty_num > 1 && dp.to[1] == SQ_NONE)
            capSq = m.to_sq();
    }
}

// modifiers

void Engine::set_numa_config_from_option(const std::string& o) {
    if (o == "auto" || o == "system")
    {
        numaContext.set_numa_config(NumaConfig::from_system());
    }
    else if (o == "hardware")
    {
        // Don't respect affinity set in the system.
        numaContext.set_numa_config(NumaConfig::from_system(false));
    }
    else if (o == "none")
    {
        numaContext.set_numa_config(NumaConfig{});
    }
    else
    {
        numaContext.set_numa_config(NumaConfig::from_string(o));
    }

    // Force reallocation of threads in case affinities need to change.
    resize_threads();
    threads.ensure_network_replicated();
}

void Engine::resize_threads() {
    threads.wait_for_search_finished();
    threads.set(numaContext.get_numa_config(), {options, threads, tt, networks}, updateContext);

    // Reallocate the hash with the new threadpool size
    set_tt_size(options["Hash"]);
    threads.ensure_network_replicated();
}

void Engine::set_tt_size(size_t mb) {
    wait_for_search_finished();
    tt.resize(mb, threads);
}

void Engine::set_ponderhit(bool b) { threads.main_manager()->ponder = b; }

// network related

void Engine::verify_networks() const {
    networks->big.verify(options["EvalFile"]);
    networks->small.verify(options["EvalFileSmall"]);
}

void Engine::load_networks() {
    networks.modify_and_replicate([this](NN::Networks& networks_) {
        networks_.big.load(binaryDirectory, options["EvalFile"]);
        networks_.small.load(binaryDirectory, options["EvalFileSmall"]);
    });
    threads.clear();
    threads.ensure_network_replicated();
}

void Engine::load_big_network(const std::string& file) {
    networks.modify_and_replicate(
      [this, &file](NN::Networks& networks_) { networks_.big.load(binaryDirectory, file); });
    threads.clear();
    threads.ensure_network_replicated();
}

void Engine::load_small_network(const std::string& file) {
    networks.modify_and_replicate(
      [this, &file](NN::Networks& networks_) { networks_.small.load(binaryDirectory, file); });
    threads.clear();
    threads.ensure_network_replicated();
}

void Engine::save_network(const std::pair<std::optional<std::string>, std::string> files[2]) {
    networks.modify_and_replicate([&files](NN::Networks& networks_) {
        networks_.big.save(files[0].first);
        networks_.small.save(files[1].first);
    });
}

// utility functions

void Engine::trace_eval() const {
    StateListPtr trace_states(new std::deque<StateInfo>(1));
    Position     p;
    p.set(pos.fen(), options["UCI_Chess960"], &trace_states->back());

    verify_networks();

    sync_cout << "\n" << Eval::trace(p, *networks) << sync_endl;
}

const OptionsMap& Engine::get_options() const { return options; }
OptionsMap&       Engine::get_options() { return options; }

std::string Engine::fen() const { return pos.fen(); }

void Engine::flip() { pos.flip(); }

std::string Engine::visualize() const {
    std::stringstream ss;
    ss << pos;
    return ss.str();
}

std::vector<std::pair<size_t, size_t>> Engine::get_bound_thread_count_by_numa_node() const {
    auto                                   counts = threads.get_bound_thread_count_by_numa_node();
    const NumaConfig&                      cfg    = numaContext.get_numa_config();
    std::vector<std::pair<size_t, size_t>> ratios;
    NumaIndex                              n = 0;
    for (; n < counts.size(); ++n)
        ratios.emplace_back(counts[n], cfg.num_cpus_in_numa_node(n));
    if (!counts.empty())
        for (; n < cfg.num_numa_nodes(); ++n)
            ratios.emplace_back(0, cfg.num_cpus_in_numa_node(n));
    return ratios;
}

std::string Engine::get_numa_config_as_string() const {
    return numaContext.get_numa_config().to_string();
}

std::string Engine::numa_config_information_as_string() const {
    auto cfgStr = get_numa_config_as_string();
    return "Available processors: " + cfgStr;
}

std::string Engine::thread_binding_information_as_string() const {
    auto              boundThreadsByNode = get_bound_thread_count_by_numa_node();
    std::stringstream ss;

    size_t threadsSize = threads.size();
    ss << "Using " << threadsSize << (threadsSize > 1 ? " threads" : " thread");

    if (boundThreadsByNode.empty())
        return ss.str();

    ss << " with NUMA node thread binding: ";

    bool isFirst = true;

    for (auto&& [current, total] : boundThreadsByNode)
    {
        if (!isFirst)
            ss << ":";
        ss << current << "/" << total;
        isFirst = false;
    }

    return ss.str();
}

}
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef _WIN32
#else
    #define WIN32_LEAN_AND_MEAN
    #ifndef NOMINMAX
        #define NOMINMAX  // Disable macros min() and max()
    #endif
#endif

using namespace Stockfish::Tablebases;

int Stockfish::Tablebases::MaxCardinality;

namespace Stockfish {

namespace {

constexpr int TBPIECES = 7;  // Max number of supported pieces
constexpr int MAX_DTZ =
  1 << 18;  // Max DTZ supported times 2, large enough to deal with the syzygy TB limit.

enum {
    BigEndian,
    LittleEndian
};
enum TBType {
    WDL,
    DTZ
};  // Used as template parameter

// Each table has a set of flags: all of them refer to DTZ tables, the last one to WDL tables
enum TBFlag {
    STM         = 1,
    Mapped      = 2,
    WinPlies    = 4,
    LossPlies   = 8,
    Wide        = 16,
    SingleValue = 128
};

inline WDLScore operator-(WDLScore d) { return WDLScore(-int(d)); }
inline Square   operator^(Square s, int i) { return Square(int(s) ^ i); }

int MapPawns[SQUARE_NB];
int MapB1H1H7[SQUARE_NB];
int MapA1D1D4[SQUARE_NB];
int MapKK[10][SQUARE_NB];  // [MapA1D1D4][SQUARE_NB]

int Binomial[6][SQUARE_NB];     // [k][n] k elements from a set of n elements
int LeadPawnIdx[6][SQUARE_NB];  // [leadPawnsCnt][SQUARE_NB]
int LeadPawnsSize[6][4];        // [leadPawnsCnt][FILE_A..FILE_D]

// Comparison function to sort leading pawns in ascending MapPawns[] order
bool pawns_comp(Square i, Square j) { return MapPawns[i] < MapPawns[j]; }
int  off_A1H8(Square sq) { return int(rank_of(sq)) - file_of(sq); }

constexpr Value WDL_to_value[] = {-VALUE_MATE + MAX_PLY + 1, VALUE_DRAW - 2, VALUE_DRAW,
                                  VALUE_DRAW + 2, VALUE_MATE - MAX_PLY - 1};

template<typename T, int Half = sizeof(T) / 2, int End = sizeof(T) - 1>
inline void swap_endian(T& x) {
    static_assert(std::is_unsigned_v<T>, "Argument of swap_endian not unsigned");

    uint8_t tmp, *c = (uint8_t*) &x;
    for (int i = 0; i < Half; ++i)
        tmp = c[i], c[i] = c[End - i], c[End - i] = tmp;
}
template<>
inline void swap_endian<uint8_t>(uint8_t&) {}

template<typename T, int LE>
T number(void* addr) {
    T v;

    if (uintptr_t(addr) & (alignof(T) - 1))  // Unaligned pointer (very rare)
        std::memcpy(&v, addr, sizeof(T));
    else
        v = *((T*) addr);

    if (LE != IsLittleEndian)
        swap_endian(v);
    return v;
}

// DTZ tables don't store valid scores for moves that reset the rule50 counter
// like captures and pawn moves but we can easily recover the correct dtz of the
// previous move if we know the position's WDL score.
int dtz_before_zeroing(WDLScore wdl) {
    return wdl == WDLWin         ? 1
         : wdl == WDLCursedWin   ? 101
         : wdl == WDLBlessedLoss ? -101
         : wdl == WDLLoss        ? -1
                                 : 0;
}

// Return the sign of a number (-1, 0, 1)
template<typename T>
int sign_of(T val) {
    return (T(0) < val) - (val < T(0));
}

// Numbers in little-endian used by sparseIndex[] to point into blockLength[]
struct SparseEntry {
    char block[4];   // Number of block
    char offset[2];  // Offset within the block
};

static_assert(sizeof(SparseEntry) == 6, "SparseEntry must be 6 bytes");

using Sym = uint16_t;  // Huffman symbol

struct LR {
    enum Side {
        Left,
        Right
    };

    uint8_t lr[3];  // The first 12 bits is the left-hand symbol, the second 12
                    // bits is the right-hand symbol. If the symbol has length 1,
                    // then the left-hand symbol is the stored value.
    template<Side S>
    Sym get() {
        return S == Left  ? ((lr[1] & 0xF) << 8) | lr[0]
             : S == Right ? (lr[2] << 4) | (lr[1] >> 4)
                          : (assert(false), Sym(-1));
    }
};

static_assert(sizeof(LR) == 3, "LR tree entry must be 3 bytes");

// Tablebases data layout is structured as following:
//
//  TBFile:   memory maps/unmaps the physical .rtbw and .rtbz files
//  TBTable:  one object for each file with corresponding indexing information
//  TBTables: has ownership of TBTable objects, keeping a list and a hash

// class TBFile memory maps/unmaps the single .rtbw and .rtbz files. Files are
// memory mapped for best performance. Files are mapped at first access: at init
// time only existence of the file is checked.
class TBFile: public std::ifstream {

    std::string fname;

   public:
    // Look for and open the file among the Paths directories where the .rtbw
    // and .rtbz files can be found. Multiple directories are separated by ";"
    // on Windows and by ":" on Unix-based operating systems.
    //
    // Example:
    // C:\tb\wdl345;C:\tb\wdl6;D:\tb\dtz345;D:\tb\dtz6
    static std::string Paths;

    TBFile(const std::string& f) {

#ifndef _WIN32
        constexpr char SepChar = ':';
#else
        constexpr char SepChar = ';';
#endif
        std::stringstream ss(Paths);
        std::string       path;

        while (std::getline(ss, path, SepChar))
        {
            fname = path + "/" + f;
            std::ifstream::open(fname);
            if (is_open())
                return;
        }
    }

    // Memory map the file and check it.
    uint8_t* map(void** baseAddress, uint64_t* mapping, TBType type) {
        if (is_open())
            close();  // Need to re-open to get native file descriptor

#ifndef _WIN32
        struct stat statbuf;
        int         fd = ::open(fname.c_str(), O_RDONLY);

        if (fd == -1)
            return *baseAddress = nullptr, nullptr;

        fstat(fd, &statbuf);

        if (statbuf.st_size % 64 != 16)
        {
            std::cerr << "Corrupt tablebase file " << fname << std::endl;
            exit(EXIT_FAILURE);
        }

        *mapping     = statbuf.st_size;
        *baseAddress = mmap(nullptr, statbuf.st_size, PROT_READ, MAP_SHARED, fd, 0);
    #if defined(MADV_RANDOM)
        madvise(*baseAddress, statbuf.st_size, MADV_RANDOM);
    #endif
        ::close(fd);

        if (*baseAddress == MAP_FAILED)
        {
            std::cerr << "Could not mmap() " << fname << std::endl;
            exit(EXIT_FAILURE);
        }
#else
        // Note FILE_FLAG_RANDOM_ACCESS is only a hint to Windows and as such may get ignored.
        HANDLE fd = CreateFileA(fname.c_str(), GENERIC_READ, FILE_SHARE_READ, nullptr,
                                OPEN_EXISTING, FILE_FLAG_RANDOM_ACCESS, nullptr);

        if (fd == INVALID_HANDLE_VALUE)
            return *baseAddress = nullptr, nullptr;

        DWORD size_high;
        DWORD size_low = GetFileSize(fd, &size_high);

        if (size_low % 64 != 16)
        {
            std::cerr << "Corrupt tablebase file " << fname << std::endl;
            exit(EXIT_FAILURE);
        }

        HANDLE mmap = CreateFileMapping(fd, nullptr, PAGE_READONLY, size_high, size_low, nullptr);
        CloseHandle(fd);

        if (!mmap)
        {
            std::cerr << "CreateFileMapping() failed" << std::endl;
            exit(EXIT_FAILURE);
        }

        *mapping     = uint64_t(mmap);
        *baseAddress = MapViewOfFile(mmap, FILE_MAP_READ, 0, 0, 0);

        if (!*baseAddress)
        {
            std::cerr << "MapViewOfFile() failed, name = " << fname
                      << ", error = " << GetLastError() << std::endl;
            exit(EXIT_FAILURE);
        }
#endif
        uint8_t* data = (uint8_t*) *baseAddress;

        constexpr uint8_t Magics[][4] = {{0xD7, 0x66, 0x0C, 0xA5}, {0x71, 0xE8, 0x23, 0x5D}};

        if (memcmp(data, Magics[type == WDL], 4))
        {
            std::cerr << "Corrupted table in file " << fname << std::endl;
            unmap(*baseAddress, *mapping);
            return *baseAddress = nullptr, nullptr;
        }

        return data + 4;  // Skip Magics's header
    }

    static void unmap(void* baseAddress, uint64_t mapping) {

#ifndef _WIN32
        munmap(baseAddress, mapping);
#else
        UnmapViewOfFile(baseAddress);
        CloseHandle((HANDLE) mapping);
#endif
    }
};

std::string TBFile::Paths;

// struct PairsData contains low-level indexing information to access TB data.
// There are 8, 4, or 2 PairsData records for each TBTable, according to the type
// of table and if positions have pawns or not. It is populated at first access.
struct PairsData {
    uint8_t   flags;            // Table flags, see enum TBFlag
    uint8_t   maxSymLen;        // Maximum length in bits of the Huffman symbols
    uint8_t   minSymLen;        // Minimum length in bits of the Huffman symbols
    uint32_t  blocksNum;        // Number of blocks in the TB file
    size_t    sizeofBlock;      // Block size in bytes
    size_t    span;             // About every span values there is a SparseIndex[] entry
    Sym*      lowestSym;        // lowestSym[l] is the symbol of length l with the lowest value
    LR*       btree;            // btree[sym] stores the left and right symbols that expand sym
    uint16_t* blockLength;      // Number of stored positions (minus one) for each block: 1..65536
    uint32_t  blockLengthSize;  // Size of blockLength[] table: padded so it's bigger than blocksNum
    SparseEntry* sparseIndex;   // Partial indices into blockLength[]
    size_t       sparseIndexSize;  // Size of SparseIndex[] table
    uint8_t*     data;             // Start of Huffman compressed data
    std::vector<uint64_t>
      base64;  // base64[l - min_sym_len] is the 64bit-padded lowest symbol of length l
    std::vector<uint8_t>
           symlen;  // Number of values (-1) represented by a given Huffman symbol: 1..256
    Piece    pieces[TBPIECES];        // Position pieces: the order of pieces defines the groups
    uint64_t groupIdx[TBPIECES + 1];  // Start index used for the encoding of the group's pieces
    int      groupLen[TBPIECES + 1];  // Number of pieces in a given group: KRKN -> (3, 1)
    uint16_t map_idx[4];              // WDLWin, WDLLoss, WDLCursedWin, WDLBlessedLoss (used in DTZ)
};

// struct TBTable contains indexing information to access the corresponding TBFile.
// There are 2 types of TBTable, corresponding to a WDL or a DTZ file. TBTable
// is populated at init time but the nested PairsData records are populated at
// first access, when the corresponding file is memory mapped.
template<TBType Type>
struct TBTable {
    using Ret = std::conditional_t<Type == WDL, WDLScore, int>;

    static constexpr int Sides = Type == WDL ? 2 : 1;

    std::atomic_bool ready;
    void*            baseAddress;
    uint8_t*         map;
    uint64_t         mapping;
    Key              key;
    Key              key2;
    int              pieceCount;
    bool             hasPawns;
    bool             hasUniquePieces;
    uint8_t          pawnCount[2];     // [Lead color / other color]
    PairsData        items[Sides][4];  // [wtm / btm][FILE_A..FILE_D or 0]

    PairsData* get(int stm, int f) { return &items[stm % Sides][hasPawns ? f : 0]; }

    TBTable() :
        ready(false),
        baseAddress(nullptr) {}
    explicit TBTable(const std::string& code);
    explicit TBTable(const TBTable<WDL>& wdl);

    ~TBTable() {
        if (baseAddress)
            TBFile::unmap(baseAddress, mapping);
    }
};

template<>
TBTable<WDL>::TBTable(const std::string& code) :
    TBTable() {

    StateInfo st;
    Position  pos;

    key        = pos.set(code, WHITE, &st).material_key();
    pieceCount = pos.count<ALL_PIECES>();
    hasPawns   = pos.pieces(PAWN);

    hasUniquePieces = false;
    for (Color c : {WHITE, BLACK})
        for (PieceType pt = PAWN; pt < KING; ++pt)
            if (popcount(pos.pieces(c, pt)) == 1)
                hasUniquePieces = true;

    // Set the leading color. In case both sides have pawns the leading color
    // is the side with fewer pawns because this leads to better compression.
    bool c = !pos.count<PAWN>(BLACK)
          || (pos.count<PAWN>(WHITE) && pos.count<PAWN>(BLACK) >= pos.count<PAWN>(WHITE));

    pawnCount[0] = pos.count<PAWN>(c ? WHITE : BLACK);
    pawnCount[1] = pos.count<PAWN>(c ? BLACK : WHITE);

    key2 = pos.set(code, BLACK, &st).material_key();
}

template<>
TBTable<DTZ>::TBTable(const TBTable<WDL>& wdl) :
    TBTable() {

    // Use the corresponding WDL table to avoid recalculating all from scratch
    key             = wdl.key;
    key2            = wdl.key2;
    pieceCount      = wdl.pieceCount;
    hasPawns        = wdl.hasPawns;
    hasUniquePieces = wdl.hasUniquePieces;
    pawnCount[0]    = wdl.pawnCount[0];
    pawnCount[1]    = wdl.pawnCount[1];
}

// class TBTables creates and keeps ownership of the TBTable objects, one for
// each TB file found. It supports a fast, hash-based, table lookup. Populated
// at init time, accessed at probe time.
class TBTables {

    struct Entry {
        Key           key;
        TBTable<WDL>* wdl;
        TBTable<DTZ>* dtz;

        template<TBType Type>
        TBTable<Type>* get() const {
            return (TBTable<Type>*) (Type == WDL ? (void*) wdl : (void*) dtz);
        }
    };

    static constexpr int Size     = 1 << 12;  // 4K table, indexed by key's 12 lsb
    static constexpr int Overflow = 1;  // Number of elements allowed to map to the last bucket

    Entry hashTable[Size + Overflow];

    std::deque<TBTable<WDL>> wdlTable;
    std::deque<TBTable<DTZ>> dtzTable;
    size_t                   foundDTZFiles = 0;
    size_t                   foundWDLFiles = 0;

    void insert(Key key, TBTable<WDL>* wdl, TBTable<DTZ>* dtz) {
        uint32_t homeBucket = uint32_t(key) & (Size - 1);
        Entry    entry{key, wdl, dtz};

        // Ensure last element is empty to avoid overflow when looking up
        for (uint32_t bucket = homeBucket; bucket < Size + Overflow - 1; ++bucket)
        {
            Key otherKey = hashTable[bucket].key;
            if (otherKey == key || !hashTable[bucket].get<WDL>())
            {
                hashTable[bucket] = entry;
                return;
            }

            // Robin Hood hashing: If we've probed for longer than this element,
            // insert here and search for a new spot for the other element instead.
            uint32_t otherHomeBucket = uint32_t(otherKey) & (Size - 1);
            if (otherHomeBucket > homeBucket)
            {
                std::swap(entry, hashTable[bucket]);
                key        = otherKey;
                homeBucket = otherHomeBucket;
            }
        }
        std::cerr << "TB hash table size too low!" << std::endl;
        exit(EXIT_FAILURE);
    }

   public:
    template<TBType Type>
    TBTable<Type>* get(Key key) {
        for (const Entry* entry = &hashTable[uint32_t(key) & (Size - 1)];; ++entry)
        {
            if (entry->key == key || !entry->get<Type>())
                return entry->get<Type>();
        }
    }

    void clear() {
        memset(hashTable, 0, sizeof(hashTable));
        wdlTable.clear();
        dtzTable.clear();
        foundDTZFiles = 0;
        foundWDLFiles = 0;
    }

    void info() const {
        sync_cout << "info string Found " << foundWDLFiles << " WDL and " << foundDTZFiles
                  << " DTZ tablebase files (up to " << MaxCardinality << "-man)." << sync_endl;
    }

    void add(const std::vector<PieceType>& pieces);
};

TBTables TBTables;

// If the corresponding file exists two new objects TBTable<WDL> and TBTable<DTZ>
// are created and added to the lists and hash table. Called at init time.
void TBTables::add(const std::vector<PieceType>& pieces) {

    std::string code;

    for (PieceType pt : pieces)
        code += PieceToChar[pt];
    code.insert(code.find('K', 1), "v");

    TBFile file_dtz(code + ".rtbz");  // KRK -> KRvK
    if (file_dtz.is_open())
    {
        file_dtz.close();
        foundDTZFiles++;
    }

    TBFile file(code + ".rtbw");  // KRK -> KRvK

    if (!file.is_open())  // Only WDL file is checked
        return;

    file.close();
    foundWDLFiles++;

    MaxCardinality = std::max(int(pieces.size()), MaxCardinality);

    wdlTable.emplace_back(code);
    dtzTable.emplace_back(wdlTable.back());

    // Insert into the hash keys for both colors: KRvK with KR white and black
    insert(wdlTable.back().key, &wdlTable.back(), &dtzTable.back());
    insert(wdlTable.back().key2, &wdlTable.back(), &dtzTable.back());
}

// TB tables are compressed with canonical Huffman code. The compressed data is divided into
// blocks of size d->sizeofBlock, and each block stores a variable number of symbols.
// Each symbol represents either a WDL or a (remapped) DTZ value, or a pair of other symbols
// (recursively). If you keep expanding the symbols in a block, you end up with up to 65536
// WDL or DTZ values. Each symbol represents up to 256 values and will correspond after
// Huffman coding to at least 1 bit. So a block of 32 bytes corresponds to at most
// 32 x 8 x 256 = 65536 values. This maximum is only reached for tables that consist mostly
// of draws or mostly of wins, but such tables are actually quite common. In principle, the
// blocks in WDL tables are 64 bytes long (and will be aligned on cache lines). But for
// mostly-draw or mostly-win tables this can leave many 64-byte blocks only half-filled, so
// in such cases blocks are 32 bytes long. The blocks of DTZ tables are up to 1024 bytes long.
// The generator picks the size that leads to the smallest table. The "book" of symbols and
// Huffman codes are the same for all blocks in the table. A non-symmetric pawnless TB file
// will have one table for wtm and one for btm, a TB file with pawns will have tables per
// file a,b,c,d also, in this case, one set for wtm and one for btm.
int decompress_pairs(PairsData* d, uint64_t idx) {

    // Special case where all table positions store the same value
    if (d->flags & TBFlag::SingleValue)
        return d->minSymLen;

    // First we need to locate the right block that stores the value at index "idx".
    // Because each block n stores blockLength[n] + 1 values, the index i of the block
    // that contains the value at position idx is:
    //
    //                    for (i = -1, sum = 0; sum <= idx; i++)
    //                        sum += blockLength[i + 1] + 1;
    //
    // This can be slow, so we use SparseIndex[] populated with a set of SparseEntry that
    // point to known indices into blockLength[]. Namely SparseIndex[k] is a SparseEntry
    // that stores the blockLength[] index and the offset within that block of the value
    // with index I(k), where:
    //
    //       I(k) = k * d->span + d->span / 2      (1)

    // First step is to get the 'k' of the I(k) nearest to our idx, using definition (1)
    uint32_t k = uint32_t(idx / d->span);

    // Then we read the corresponding SparseIndex[] entry
    uint32_t block  = number<uint32_t, LittleEndian>(&d->sparseIndex[k].block);
    int       offset = number<uint16_t, LittleEndian>(&d->sparseIndex[k].offset);

    // Now compute the difference idx - I(k). From the definition of k, we know that
    //
    //       idx = k * d->span + idx % d->span    (2)
    //
    // So from (1) and (2) we can compute idx - I(K):
    int diff = idx % d->span - d->span / 2;

    // Sum the above to offset to find the offset corresponding to our idx
    offset += diff;

    // Move to the previous/next block, until we reach the correct block that contains idx,
    // that is when 0 <= offset <= d->blockLength[block]
    while (offset < 0)
        offset += d->blockLength[--block] + 1;

    while (offset > d->blockLength[block])
        offset -= d->blockLength[block++] + 1;

    // Finally, we find the start address of our block of canonical Huffman symbols
    uint32_t* ptr = (uint32_t*)(d->data + (uint64_t(block) * d->sizeofBlock));

    // Read the first 64 bits in our block, this is a (truncated) sequence of
    // unknown number of symbols of unknown length but we know the first one
    // is at the beginning of this 64-bit sequence.
    uint64_t buf64 = number<uint64_t, BigEndian>(ptr);
    ptr += 2;
    int buf64Size = 64;
    Sym sym;

    while (true)
    {
        int len = 0; // This is the symbol length - d->min_sym_len

        // Now get the symbol length. For any symbol s64 of length l right-padded
        // to 64 bits we know that d->base64[l-1] >= s64 >= d->base64[l] so we
        // can find the symbol length iterating through base64[].
        while (buf64 < d->base64[len])
            ++len;

        // All the symbols of a given length are consecutive integers (numerical
        // sequence property), so we can compute the offset of our symbol of
        // length len, stored at the beginning of buf64.
        sym = Sym((buf64 - d->base64[len]) >> (64 - len - d->minSymLen));

        // Now add the value of the lowest symbol of length len to get our symbol
        sym += number<Sym, LittleEndian>(&d->lowestSym[len]);

        // If our offset is within the number of values represented by symbol sym,
        // we are done.
        if (offset < d->symlen[sym] + 1)
            break;

        // ...otherwise update the offset and continue to iterate
        offset -= d->symlen[sym] + 1;
        len += d->minSymLen;  // Get the real length
        buf64 <<= len;        // Consume the just processed symbol
        buf64Size -= len;

        if (buf64Size <= 32)
        {   // Refill the buffer
            buf64Size += 32;
            buf64 |= uint64_t(number<uint32_t, BigEndian>(ptr++)) << (64 - buf64Size);
        }
    }

    // Now we have our symbol that expands into d->symlen[sym] + 1 symbols.
    // We binary-search for our value recursively expanding into the left and
    // right child symbols until we reach a leaf node where symlen[sym] + 1 == 1
    // that will store the value we need.
    while (d->symlen[sym])
    {
        Sym left = d->btree[sym].get<LR::Left>();

        // If a symbol contains 36 sub-symbols (d->symlen[sym] + 1 = 36) and
        // expands in a pair (d->symlen[left] = 23, d->symlen[right] = 11), then
        // we know that, for instance, the tenth value (offset = 10) will be on
        // the left side because in Recursive Pairing child symbols are adjacent.
        if (offset < d->symlen[left] + 1)
            sym = left;
        else
        {
            offset -= d->symlen[left] + 1;
            sym = d->btree[sym].get<LR::Right>();
        }
    }

    return d->btree[sym].get<LR::Left>();
}

bool check_dtz_stm(TBTable<WDL>*, int, File) { return true; }

bool check_dtz_stm(TBTable<DTZ>* entry, int stm, File f) {

    auto flags = entry->get(stm, f)->flags;
    return   (flags & TBFlag::STM) == stm || ((entry->key == entry->key2) && !entry->hasPawns);
}

// DTZ scores are sorted by frequency of occurrence and then assigned the
// values 0, 1, 2, ... in order of decreasing frequency. This is done for each
// of the four WDLScore values. The mapping information necessary to reconstruct
// the original values are stored in the TB file and read during map[] init.
WDLScore map_score(TBTable<WDL>*, File, int value, WDLScore) { return WDLScore(value - 2); }

int map_score(TBTable<DTZ>* entry, File f, int value, WDLScore wdl) {

    constexpr int WDLMap[] = { 1, 3, 0, 2, 0 };

    auto flags = entry->get(0, f)->flags;

    uint8_t*  map = entry->map;
    uint16_t* idx = entry->get(0, f)->map_idx;
    if (flags & TBFlag::Mapped) {
        if (flags & TBFlag::Wide)
            value = ((uint16_t *)map)[idx[WDLMap[wdl + 2]] + value];
        else
            value = map[idx[WDLMap[wdl + 2]] + value];
    }

    // DTZ tables store distance to zero in number of moves or plies. We
    // want to return plies, so we have to convert to plies when needed.
    if (   (wdl == WDLWin  && !(flags & TBFlag::WinPlies))
        || (wdl == WDLLoss && !(flags & TBFlag::LossPlies))
        ||  wdl == WDLCursedWin
        ||  wdl == WDLBlessedLoss)
        value *= 2;

    return value + 1;
}

// A temporary fix for the compiler bug with AVX-512. (#4450)
#ifdef USE_AVX512
  #if defined(__clang__) && defined(__clang_major__) && __clang_major__ >= 15
    #define CLANG_AVX512_BUG_FIX __attribute__((optnone))
  #endif
#endif

#ifndef CLANG_AVX512_BUG_FIX
    #define CLANG_AVX512_BUG_FIX
#endif

// Compute a unique index out of a position and use it to probe the TB file. To
// encode k pieces of the same type and color, first sort the pieces by square in
// ascending order s1 <= s2 <= ... <= sk then compute the unique index as:
//
//      idx = Binomial[1][s1] + Binomial[2][s2] + ... + Binomial[k][sk]
//
template<typename T, typename Ret = typename T::Ret>
CLANG_AVX512_BUG_FIX Ret
do_probe_table(const Position& pos, T* entry, WDLScore wdl, ProbeState* result) {

    Square     squares[TBPIECES];
    Piece      pieces[TBPIECES];
    uint64_t   idx;
    int        next = 0, size = 0, leadPawnsCnt = 0;
    PairsData* d;
    Bitboard   b, leadPawns = 0;
    File       tbFile = FILE_A;

    // A given TB entry like KRK has associated two material keys: KRvK and KvKR.
    // If both sides have the same pieces keys are equal. In this case TB tables
    // only stores the 'white to move' case, so if the position to lookup has black
    // to move, we need to switch the color and flip the squares before to lookup.
    bool symmetricBlackToMove = (entry->key == entry->key2 && pos.side_to_move());

    // TB files are calculated for white as the stronger side. For instance, we
    // have KRvK, not KvKR. A position where the stronger side is white will have
    // its material key == entry->key, otherwise we have to switch the color and
    // flip the squares before to lookup.
    bool blackStronger = (pos.material_key() != entry->key);

    int flipColor   = (symmetricBlackToMove || blackStronger) * 8;
    int flipSquares = (symmetricBlackToMove || blackStronger) * 56;
    int stm         = (symmetricBlackToMove || blackStronger) ^ pos.side_to_move();

    // For pawns, TB files store 4 separate tables according if leading pawn is on
    // file a, b, c or d after reordering. The leading pawn is the one with maximum
    // MapPawns[] value, that is the one most toward the edges and with lowest rank.
    if (entry->hasPawns)
    {
        // In all the 4 tables, pawns are at the beginning of the piece sequence and
        // their color is the reference one. So we just pick the first one.
        Piece pc = Piece(entry->get(0, 0)->pieces[0] ^ flipColor);

        assert(type_of(pc) == PAWN);

        leadPawns = b = pos.pieces(color_of(pc), PAWN);
        do
            squares[size++] = pop_lsb(b) ^ flipSquares;
        while (b);

        leadPawnsCnt = size;

        std::swap(squares[0], *std::max_element(squares, squares + leadPawnsCnt, pawns_comp));

        tbFile = File(edge_distance(file_of(squares[0])));
    }

    // DTZ tables are one-sided, i.e. they store positions only for white to
    // move or only for black to move, so check for side to move to be stm,
    // early exit otherwise.
    if (!check_dtz_stm(entry, stm, tbFile))
        return *result = CHANGE_STM, Ret();

    // Now we are ready to get all the position pieces (but the lead pawns) and
    // directly map them to the correct color and square.
    b = pos.pieces() ^ leadPawns;
    do
    {
        Square s       = pop_lsb(b);
        squares[size]  = s ^ flipSquares;
        pieces[size++] = Piece(pos.piece_on(s) ^ flipColor);
    } while (b);

    assert(size >= 2);

    d = entry->get(stm, tbFile);

    // Then we reorder the pieces to have the same sequence as the one stored
    // in pieces[i]: the sequence that ensures the best compression.
    for (int i = leadPawnsCnt; i < size - 1; ++i)
        for (int j = i + 1; j < size; ++j)
            if (d->pieces[i] == pieces[j])
            {
                std::swap(pieces[i], pieces[j]);
                std::swap(squares[i], squares[j]);
                break;
            }

    // Now we map again the squares so that the square of the lead piece is in
    // the triangle A1-D1-D4.
    if (file_of(squares[0]) > FILE_D)
        for (int i = 0; i < size; ++i)
            squares[i] = flip_file(squares[i]);

    // Encode leading pawns starting with the one with minimum MapPawns[] and
    // proceeding in ascending order.
    if (entry->hasPawns)
    {
        idx = LeadPawnIdx[leadPawnsCnt][squares[0]];

        std::stable_sort(squares + 1, squares + leadPawnsCnt, pawns_comp);

        for (int i = 1; i < leadPawnsCnt; ++i)
            idx += Binomial[i][MapPawns[squares[i]]];

        goto encode_remaining; // With pawns we have finished special treatments
    }

    // In positions without pawns, we further flip the squares to ensure leading
    // piece is below RANK_5.
    if (rank_of(squares[0]) > RANK_4)
        for (int i = 0; i < size; ++i)
            squares[i] = flip_rank(squares[i]);

    // Look for the first piece of the leading group not on the A1-D4 diagonal
    // and ensure it is mapped below the diagonal.
    for (int i = 0; i < d->groupLen[0]; ++i)
    {
        if (!off_A1H8(squares[i]))
            continue;

        if (off_A1H8(squares[i]) > 0) // A1-H8 diagonal flip: SQ_A3 -> SQ_C1
            for (int j = i; j < size; ++j)
                squares[j] = Square(((squares[j] >> 3) | (squares[j] << 3)) & 63);
        break;
    }

    // Encode the leading group.
    //
    // Suppose we have KRvK. Let's say the pieces are on square numbers wK, wR
    // and bK (each 0...63). The simplest way to map this position to an index
    // is like this:
    //
    //   index = wK * 64 * 64 + wR * 64 + bK;
    //
    // But this way the TB is going to have 64*64*64 = 262144 positions, with
    // lots of positions being equivalent (because they are mirrors of each
    // other) and lots of positions being invalid (two pieces on one square,
    // adjacent kings, etc.).
    // Usually the first step is to take the wK and bK together. There are just
    // 462 ways legal and not-mirrored ways to place the wK and bK on the board.
    // Once we have placed the wK and bK, there are 62 squares left for the wR
    // Mapping its square from 0..63 to available squares 0..61 can be done like:
    //
    //   wR -= (wR > wK) + (wR > bK);
    //
    // In words: if wR "comes later" than wK, we deduct 1, and the same if wR
    // "comes later" than bK. In case of two same pieces like KRRvK we want to
    // place the two Rs "together". If we have 62 squares left, we can place two
    // Rs "together" in 62 * 61 / 2 ways (we divide by 2 because rooks can be
    // swapped and still get the same position.)
    //
    // In case we have at least 3 unique pieces (including kings) we encode them
    // together.
    if (entry->hasUniquePieces)
    {
        int adjust1 =  squares[1] > squares[0];
        int adjust2 = (squares[2] > squares[0]) + (squares[2] > squares[1]);

        // First piece is below a1-h8 diagonal. MapA1D1D4[] maps the b1-d1-d3
        // triangle to 0...5. There are 63 squares for second piece and 62
        // (mapped to 0...61) for the third.
        if (off_A1H8(squares[0]))
            idx = (   MapA1D1D4[squares[0]]  * 63
                   + (squares[1] - adjust1)) * 62
                   +  squares[2] - adjust2;

        // First piece is on a1-h8 diagonal, second below: map this occurrence to
        // 6 to differentiate from the above case, rank_of() maps a1-d4 diagonal
        // to 0...3 and finally MapB1H1H7[] maps the b1-h1-h7 triangle to 0..27.
        else if (off_A1H8(squares[1]))
            idx = (  6 * 63 + rank_of(squares[0]) * 28
                   + MapB1H1H7[squares[1]])       * 62
                   + squares[2] - adjust2;

        // First two pieces are on a1-h8 diagonal, third below
        else if (off_A1H8(squares[2]))
            idx =  6 * 63 * 62 + 4 * 28 * 62
                 +  rank_of(squares[0])        * 7 * 28
                 + (rank_of(squares[1]) - adjust1) * 28
                 +  MapB1H1H7[squares[2]];

        // All 3 pieces on the diagonal a1-h8
        else
            idx = 6 * 63 * 62 + 4 * 28 * 62 + 4 * 7 * 28
                 +  rank_of(squares[0])         * 7 * 6
                 + (rank_of(squares[1]) - adjust1)  * 6
                 + (rank_of(squares[2]) - adjust2);
    } else
        // We don't have at least 3 unique pieces, like in KRRvKBB, just map
        // the kings.
        idx = MapKK[MapA1D1D4[squares[0]]][squares[1]];

encode_remaining:
    idx *= d->groupIdx[0];
    Square* groupSq = squares + d->groupLen[0];

    // Encode remainig pawns and then pieces according to square, in ascending order
    bool remainingPawns = entry->hasPawns && entry->pawnCount[1];

    while (d->groupLen[++next])
    {
        std::stable_sort(groupSq, groupSq + d->groupLen[next]);
        uint64_t n = 0;

        // Map down a square if "comes later" than a square in the previous
        // groups (similar to what was done earlier for leading group pieces).
        for (int i = 0; i < d->groupLen[next]; ++i)
        {
            auto f      = [&](Square s) { return groupSq[i] > s; };
            auto adjust = std::count_if(squares, groupSq, f);
            n += Binomial[i + 1][groupSq[i] - adjust - 8 * remainingPawns];
        }

        remainingPawns = false;
        idx += n * d->groupIdx[next];
        groupSq += d->groupLen[next];
    }

    // Now that we have the index, decompress the pair and get the score
    return map_score(entry, tbFile, decompress_pairs(d, idx), wdl);
}

// Group together pieces that will be encoded together. The general rule is that
// a group contains pieces of the same type and color. The exception is the leading
// group that, in case of positions without pawns, can be formed by 3 different
// pieces (default) or by the king pair when there is not a unique piece apart
// from the kings. When there are pawns, pawns are always first in pieces[].
//
// As example KRKN -> KRK + N, KNNK -> KK + NN, KPPKP -> P + PP + K + K
//
// The actual grouping depends on the TB generator and can be inferred from the
// sequence of pieces in piece[] array.
template<typename T>
void set_groups(T& e, PairsData* d, int order[], File f) {

    int n = 0, firstLen = e.hasPawns ? 0 : e.hasUniquePieces ? 3 : 2;
    d->groupLen[n] = 1;

    // Number of pieces per group is stored in groupLen[], for instance in KRKN
    // the encoder will default on '111', so groupLen[] will be (3, 1).
    for (int i = 1; i < e.pieceCount; ++i)
        if (--firstLen > 0 || d->pieces[i] == d->pieces[i - 1])
            d->groupLen[n]++;
        else
            d->groupLen[++n] = 1;

    d->groupLen[++n] = 0; // Zero-terminated

    // The sequence in pieces[] defines the groups, but not the order in which
    // they are encoded. If the pieces in a group g can be combined on the board
    // in N(g) different ways, then the position encoding will be of the form:
    //
    //           g1 * N(g2) * N(g3) + g2 * N(g3) + g3
    //
    // This ensures unique encoding for the whole position. The order of the
    // groups is a per-table parameter and could not follow the canonical leading
    // pawns/pieces -> remaining pawns -> remaining pieces. In particular the
    // first group is at order[0] position and the remaining pawns, when present,
    // are at order[1] position.
    bool pp = e.hasPawns && e.pawnCount[1]; // Pawns on both sides
    int next = pp ? 2 : 1;
    int freeSquares = 64 - d->groupLen[0] - (pp ? d->groupLen[1] : 0);
    uint64_t idx = 1;

    for (int k = 0; next < n || k == order[0] || k == order[1]; ++k)
        if (k == order[0]) // Leading pawns or pieces
        {
            d->groupIdx[0] = idx;
            idx *= e.hasPawns ? LeadPawnsSize[d->groupLen[0]][f] : e.hasUniquePieces ? 31332 : 462;
        }
        else if (k == order[1]) // Remaining pawns
        {
            d->groupIdx[1] = idx;
            idx *= Binomial[d->groupLen[1]][48 - d->groupLen[0]];
        }
        else // Remaining pieces
        {
            d->groupIdx[next] = idx;
            idx *= Binomial[d->groupLen[next]][freeSquares];
            freeSquares -= d->groupLen[next++];
        }

    d->groupIdx[n] = idx;
}

// In Recursive Pairing each symbol represents a pair of children symbols. So
// read d->btree[] symbols data and expand each one in his left and right child
// symbol until reaching the leaves that represent the symbol value.
uint8_t set_symlen(PairsData* d, Sym s, std::vector<bool>& visited) {

    visited[s] = true; // We can set it now because tree is acyclic
    Sym sr = d->btree[s].get<LR::Right>();

    if (sr == 0xFFF)
        return 0;

    Sym sl = d->btree[s].get<LR::Left>();

    if (!visited[sl])
        d->symlen[sl] = set_symlen(d, sl, visited);

    if (!visited[sr])
        d->symlen[sr] = set_symlen(d, sr, visited);

    return d->symlen[sl] + d->symlen[sr] + 1;
}

uint8_t* set_sizes(PairsData* d, uint8_t* data) {

    d->flags = *data++;

    if (d->flags & TBFlag::SingleValue) {
        d->numBlocks = d->span =
        d->blocksNum = d->sparseIndexSize = 0; // Broken MSVC zero-init
        d->minSymLen = *data++; // Here we store the single value
        return data;
    }

    // groupLen[] is a zero-terminated list of group lengths, the last groupIdx[]
    // element stores the biggest index that is the tb size.
    uint64_t tbSize = d->groupIdx[std::find(d->groupLen, d->groupLen + 7, 0) - d->groupLen];

    d->sizeofBlock     = 1ULL << *data++;
    d->span            = 1ULL << *data++;
    d->sparseIndexSize = size_t((tbSize + d->span - 1) / d->span); // Round up
    auto padding       = number<uint8_t, LittleEndian>(data++);
    d->blocksNum       = number<uint32_t, LittleEndian>(data); data += sizeof(uint32_t);
    d->blockLengthSize = d->blocksNum + padding; // Padded to ensure SparseIndex[]
                                                 // does not point out of range.
    d->maxSymLen = *data++;
    d->minSymLen = *data++;
    d->lowestSym = (Sym*)data;
    d->base64.resize(d->maxSymLen - d->minSymLen + 1);

    // See https://en.wikipedia.org/wiki/Huffman_coding
    // The canonical code is ordered such that longer symbols (in terms of
    // the number of bits of their Huffman code) have a lower numeric value,
    // so that d->lowestSym[i] >= d->lowestSym[i+1] (when read as LittleEndian).
    // Starting from this we compute a base64[] table indexed by symbol length
    // and containing 64 bit values so that d->base64[i] >= d->base64[i+1].

    // Implementation note: we first cast the unsigned size_t "base64.size()"
    // to a signed int "base64_size" variable and then we are able to subtract 2,
    // avoiding unsigned overflow warnings.

    int base64_size = static_cast<int>(d->base64.size());
    for (int i = base64_size - 2; i >= 0; --i) {
        d->base64[i] = (d->base64[i + 1] + number<Sym, LittleEndian>(&d->lowestSym[i])
                                         - number<Sym, LittleEndian>(&d->lowestSym[i + 1])) / 2;

        assert(d->base64[i] * 2 >= d->base64[i+1]);
    }

    // Now left-shift by an amount so that d->base64[i] gets shifted 1 bit more
    // than d->base64[i+1] and given the above assert condition, we ensure that
    // d->base64[i] >= d->base64[i+1]. Moreover for any symbol s64 of length i
    // and right-padded to 64 bits holds d->base64[i-1] >= s64 >= d->base64[i].
    for (int i = 0; i < base64_size; ++i)
        d->base64[i] <<= 64 - i - d->minSymLen; // Right-padding to 64 bits

    data += base64_size * sizeof(Sym);
    d->symlen.resize(number<uint16_t, LittleEndian>(data));
    data += sizeof(uint16_t);
    d->btree = (LR*)data;

    // The compression scheme used is "Recursive Pairing", that replaces the most
    // frequent adjacent pair of symbols in the source message by a new symbol,
    // reevaluating the frequencies of all of the symbol pairs with respect to
    // the extended alphabet, and then repeating the process.
    // See https://web.archive.org/web/20201106232444/http://www.larsson.dogma.net/dcc99.pdf
    std::vector<bool> visited(d->symlen.size());

    for (std::size_t sym = 0; sym < d->symlen.size(); ++sym)
        if (!visited[sym])
            d->symlen[sym] = set_symlen(d, sym, visited);

    return data + d->symlen.size() * sizeof(LR) + (d->symlen.size() & 1);
}

uint8_t* set_dtz_map(TBTable<WDL>&, uint8_t* data, File) { return data; }

uint8_t* set_dtz_map(TBTable<DTZ>& e, uint8_t* data, File maxFile) {

    e.map = data;

    for (File f = FILE_A; f <= maxFile; ++f) {
        auto flags = e.get(0, f)->flags;
        if (flags & TBFlag::Mapped) {
            if (flags & TBFlag::Wide) {
                data += uintptr_t(data) & 1;  // Word alignment, we may have a mixed table
                for (int i = 0; i < 4; ++i) { // Sequence like 3,x,x,x,1,x,0,2,x,x
                    e.get(0, f)->map_idx[i] = uint16_t((uint16_t *)data - (uint16_t *) e.map + 1);
                    data += 2 * number<uint16_t, LittleEndian>(data) + 2;
                }
            }
            else {
                for (int i = 0; i < 4; ++i) {
                    e.get(0, f)->map_idx[i] = uint16_t(data - e.map + 1);
                    data += *data + 1;
                }
            }
        }
    }

    return data += uintptr_t(data) & 1; // Word alignment
}

// Populate entry's PairsData records with data from the just memory-mapped file.
// Called at first access.
template<typename T>
void set(T& e, uint8_t* data) {

    PairsData* d;

    enum {
        Split    = 1,
        HasPawns = 2
    };

    assert(e.hasPawns        == bool(*data & HasPawns));
    assert((e.key != e.key2) == bool(*data & Split));

    data++; // First byte stores flags

    const int sides  = T::Sides == 2 && (e.key != e.key2) ? 2 : 1;
    const File maxFile = e.hasPawns ? FILE_D : FILE_A;

    bool pp = e.hasPawns && e.pawnCount[1]; // Pawns on both sides

    assert(!pp || e.pawnCount[0]);

    for (File f = FILE_A; f <= maxFile; ++f) {

        for (int i = 0; i < sides; i++)
            *e.get(i, f) = PairsData();

        int order[][2] = { { *data & 0xF, pp ? *(data + 1) & 0xF : 0xF },
                           { *data >> 4, pp ? *(data + 1) >> 4 : 0xF } };
        data += 1 + pp;

        for (int k = 0; k < e.pieceCount; ++k, ++data)
            for (int i = 0; i < sides; i++)
                e.get(i, f)->pieces[k] = Piece(i ? *data >> 4 : *data & 0xF);

        for (int i = 0; i < sides; ++i)
            set_groups(e, e.get(i, f), order[i], f);
    }

    data += uintptr_t(data) & 1; // Word alignment

    for (File f = FILE_A; f <= maxFile; ++f)
        for (int i = 0; i < sides; i++)
            data = set_sizes(e.get(i, f), data);

    data = set_dtz_map(e, data, maxFile);

    for (File f = FILE_A; f <= maxFile; ++f)
        for (int i = 0; i < sides; i++)
        {
            (d = e.get(i, f))->sparseIndex = (SparseEntry*)data;
            data += d->sparseIndexSize * sizeof(SparseEntry);
        }

    for (File f = FILE_A; f <= maxFile; ++f)
        for (int i = 0; i < sides; i++)
        {
            (d = e.get(i, f))->blockLength = (uint16_t*)data;
            data += d->blockLengthSize * sizeof(uint16_t);
        }

    for (File f = FILE_A; f <= maxFile; ++f)
        for (int i = 0; i < sides; i++)
        {
            data = (uint8_t*)((uintptr_t(data) + 0x3F) & ~0x3F); // 64 byte alignment
            (d = e.get(i, f))->data = data;
            data += d->blocksNum * d->sizeofBlock;
        }
}

// If the TB file corresponding to the given position is already memory-mapped
// then return its base address, otherwise, try to memory map and init it. Called
// at every probe, memory map, and init only at first access. Function is thread
// safe and can be called concurrently.
template<TBType Type>
void* mapped(TBTable<Type>& e, const Position& pos) {

    static std::mutex mutex;

    // Use 'acquire' to avoid a thread reading 'ready' == true while
    // another is still working. (compiler reordering may cause this).
    if (e.ready.load(std::memory_order_acquire))
        return e.baseAddress; // Could be nullptr if file does not exist

    std::scoped_lock<std::mutex> lk(mutex);

    if (e.ready.load(std::memory_order_relaxed)) // Recheck under lock
        return e.baseAddress;

    // Pieces strings in decreasing order for each color, like ("KPP","KR")
    std::string fname, w, b;
    for (PieceType pt = KING; pt >= PAWN; --pt)
    {
        w += std::string(popcount(pos.pieces(WHITE, pt)), PieceToChar[pt]);
        b += std::string(popcount(pos.pieces(BLACK, pt)), PieceToChar[pt]);
    }

    fname =  (e.key == pos.material_key() ? w + 'v' + b : b + 'v' + w)
           + (Type == WDL ? ".rtbw" : ".rtbz");

    uint8_t* data = TBFile(fname).map(&e.baseAddress, &e.mapping, Type);

    if (data)
        set(e, data);

    e.ready.store(true, std::memory_order_release);
    return e.baseAddress;
}

template<TBType Type, typename Ret = typename TBTable<Type>::Ret>
Ret probe_table(const Position& pos, ProbeState* result, WDLScore wdl = WDLDraw) {

    if (pos.count<ALL_PIECES>() == 2) // KvK
        return Ret(WDLDraw);

    TBTable<Type>* entry = TBTables.get<Type>(pos.material_key());

    if (!entry || !mapped(*entry, pos))
        return *result = FAIL, Ret();

    return do_probe_table(pos, entry, wdl, result);
}

// For a position where the side to move has a winning capture it is not necessary
// to store a winning value so the generator treats such positions as "don't care"
// and tries to assign to it a value that improves the compression ratio. Similarly,
// if the side to move has a drawing capture, then the position is at least drawn.
// If the position is won, then the TB needs to store a win value. But if the
// position is drawn, the TB may store a loss value if that is better for compression.
// All of this means that during probing, the engine must look at captures and probe
// their results and must probe the position itself. The "best" result of these
// probes is the correct result for the position.
// DTZ tables do not store values when a following move is a zeroing winning move
// (winning capture or winning pawn move). Also DTZ store wrong values for positions
// where the best move is an ep-move (even if losing). So in all these cases set
// the state to ZEROING_BEST_MOVE.
template<bool CheckZeroingMoves>
WDLScore search(Position& pos, ProbeState* result) {

    WDLScore value, bestValue = WDLLoss;
    StateInfo st;

    auto moveList = MoveList<LEGAL>(pos);
    size_t totalCount = moveList.size(), moveCount = 0;

    for (const Move move : moveList)
    {
        if (   !pos.capture(move)
            && (!CheckZeroingMoves || type_of(pos.moved_piece(move)) != PAWN))
            continue;

        moveCount++;

        pos.do_move(move, st);
        value = -search<false>(pos, result);
        pos.undo_move(move);

        if (*result == FAIL)
            return WDLDraw;

        if (value > bestValue)
        {
            bestValue = value;

            if (value >= WDLWin)
            {
                *result = ZEROING_BEST_MOVE; // Winning DTZ-zeroing move
                return value;
            }
        }
    }

    // In case we have already searched all the legal moves we don't have to probe
    // the TB because the stored score could be wrong. For instance TB tables
    // do not contain information on position with ep rights, so in this case
    // the result of probe_wdl_table is wrong. Also in case of only capture
    // moves, for instance here 4K3/4q3/6p1/2k5/6p1/8/8/8 w - - 0 7, we have to
    // return with ZEROING_BEST_MOVE set.
    bool noMoreMoves = (moveCount && moveCount == totalCount);

    if (noMoreMoves)
        value = bestValue;
    else
    {
        value = probe_table<WDL>(pos, result);

        if (*result == FAIL)
            return WDLDraw;
    }

    // DTZ stores a "don't care" value if bestValue is a win
    if (bestValue >= value)
        return *result = (bestValue > WDLDraw || noMoreMoves ? ZEROING_BEST_MOVE : OK), bestValue;

    return *result = OK, value;
}

} // namespace

// Called at startup and after every change to
// "SyzygyPath" UCI option to (re)create the various tables. It is not thread
// safe, nor it needs to be.
void Tablebases::init(const std::string& paths) {

    TBTables.clear();
    MaxCardinality = 0;
    TBFile::Paths = paths;

    if (paths.empty())
        return;

    // MapB1H1H7[] encodes a square below a1-h8 diagonal to 0..27
    int code = 0;
    for (Square s = SQ_A1; s <= SQ_H8; ++s)
        if (off_A1H8(s) < 0)
            MapB1H1H7[s] = code++;

    // MapA1D1D4[] encodes a square in the a1-d1-d4 triangle to 0..9
    std::vector<Square> diagonal;
    code = 0;
    for (Square s = SQ_A1; s <= SQ_D4; ++s)
        if (off_A1H8(s) < 0 && file_of(s) <= FILE_D)
            MapA1D1D4[s] = code++;

        else if (!off_A1H8(s) && file_of(s) <= FILE_D)
            diagonal.push_back(s);

    // Diagonal squares are encoded as last ones
    for (auto s : diagonal)
        MapA1D1D4[s] = code++;

    // MapKK[] encodes all the 462 possible legal positions of two kings where
    // the first is in the a1-d1-d4 triangle. If the first king is on the a1-d4
    // diagonal, the other one shall not be above the a1-h8 diagonal.
    std::vector<std::pair<int, Square>> bothOnDiagonal;
    code = 0;
    for (int idx = 0; idx < 10; idx++)
        for (Square s1 = SQ_A1; s1 <= SQ_D4; ++s1)
            if (MapA1D1D4[s1] == idx && (idx || s1 == SQ_B1)) // SQ_B1 is mapped to 0
            {
                for (Square s2 = SQ_A1; s2 <= SQ_H8; ++s2)
                    if ((PseudoAttacks[KING][s1] | s1) & s2)
                        continue; // Illegal position

                    else if (!off_A1H8(s1) && off_A1H8(s2) > 0)
                        continue; // First on diagonal, second above

                    else if (!off_A1H8(s1) && !off_A1H8(s2))
                        bothOnDiagonal.emplace_back(idx, s2);

                    else
                        MapKK[idx][s2] = code++;
            }

    // Legal positions with both kings on a diagonal are encoded as last ones
    for (auto p : bothOnDiagonal)
        MapKK[p.first][p.second] = code++;

    // Binomial[] stores the Binomial Coefficients using Pascal rule. There
    // are Binomial[k][n] ways to choose k elements from a set of n elements.
    Binomial[0][0] = 1;

    for (int n = 1; n < 64; n++) // Squares
        for (int k = 0; k < 6 && k <= n; ++k) // Pieces
            Binomial[k][n] =  (k > 0 ? Binomial[k - 1][n - 1] : 0)
                            + (k < n ? Binomial[k    ][n - 1] : 0);

    // MapPawns[s] encodes squares a2-h7 to 0..47. This is the number of possible
    // available squares when the leading one is in 's'. Moreover the pawn with
    // highest MapPawns[] is the leading pawn, the one nearest the edge, and
    // among pawns with the same file, the one with the lowest rank.
    int availableSquares = 47; // Available squares when lead pawn is in a2

    // Init the tables for the encoding of leading pawns group: with 7-men TB we
    // can have up to 5 leading pawns (KPPPPPK).
    for (int leadPawnsCnt = 1; leadPawnsCnt <= 5; ++leadPawnsCnt)
        for (File f = FILE_A; f <= FILE_D; ++f)
        {
            // Restart the index at every file because TB table is split
            // by file, so we can reuse the same index for different files.
            int idx = 0;

            // Sum all possible combinations for a given file, starting with
            // the leading pawn on rank 2 and increasing the rank.
            for (Rank r = RANK_2; r <= RANK_7; ++r)
            {
                Square sq = make_square(f, r);

                // Compute MapPawns[] at first pass.
                // If sq is the leading pawn square, any other pawn cannot be
                // below or more toward the edge of sq. There are 47 available
                // squares when sq = a2 and reduced by 2 for any rank increase
                // due to mirroring: sq == a3 -> no a2, h2, so MapPawns[a3] = 45
                if (leadPawnsCnt == 1)
                {
                    MapPawns[sq]            = availableSquares--;
                    MapPawns[flip_file(sq)] = availableSquares--;
                }
                LeadPawnIdx[leadPawnsCnt][sq] = idx;
                idx += Binomial[leadPawnsCnt - 1][MapPawns[sq]];
            }
            // After a file is traversed, store the cumulated per-file index
            LeadPawnsSize[leadPawnsCnt][f] = idx;
        }

    // Add entries in TB tables if the corresponding ".rtbw" file exists
    for (PieceType p1 = PAWN; p1 < KING; ++p1)
    {
        TBTables.add({KING, p1, KING});

        for (PieceType p2 = PAWN; p2 <= p1; ++p2)
        {
            TBTables.add({KING, p1, p2, KING});
            TBTables.add({KING, p1, KING, p2});

            for (PieceType p3 = PAWN; p3 < KING; ++p3)
                TBTables.add({KING, p1, p2, KING, p3});

            for (PieceType p3 = PAWN; p3 <= p2; ++p3)
            {
                TBTables.add({KING, p1, p2, p3, KING});

                for (PieceType p4 = PAWN; p4 <= p3; ++p4)
                {
                    TBTables.add({KING, p1, p2, p3, p4, KING});

                    for (PieceType p5 = PAWN; p5 <= p4; ++p5)
                        TBTables.add({KING, p1, p2, p3, p4, p5, KING});

                    for (PieceType p5 = PAWN; p5 < KING; ++p5)
                        TBTables.add({KING, p1, p2, p3, p4, KING, p5});
                }

                for (PieceType p4 = PAWN; p4 < KING; ++p4)
                {
                    TBTables.add({KING, p1, p2, p3, KING, p4});

                    for (PieceType p5 = PAWN; p5 <= p4; ++p5)
                        TBTables.add({KING, p1, p2, p3, KING, p4, p5});
                }
            }

            for (PieceType p3 = PAWN; p3 <= p1; ++p3)
                for (PieceType p4 = PAWN; p4 <= (p1 == p3 ? p2 : p3); ++p4)
                    TBTables.add({KING, p1, p2, KING, p3, p4});
        }
    }

    TBTables.info();
}

// Probe the WDL table for a particular position.
// If *result != FAIL, the probe was successful.
// The return value is from the point of view of the side to move:
// -2 : loss
// -1 : loss, but draw under 50-move rule
//  0 : draw
//  1 : win, but draw under 50-move rule
//  2 : win
WDLScore Tablebases::probe_wdl(Position& pos, ProbeState* result) {

    *result = OK;
    return search<false>(pos, result);
}

// Probe the DTZ table for a particular position.
// If *result != FAIL, the probe was successful.
// The return value is from the point of view of the side to move:
//         n < -100 : loss, but draw under 50-move rule
// -100 <= n < -1   : loss in n ply (assuming 50-move counter == 0)
//        -1        : loss, the side to move is mated
//         0        : draw
//     1 < n <= 100 : win in n ply (assuming 50-move counter == 0)
//   100 < n        : win, but draw under 50-move rule
//
// The return value n can be off by 1: a return value -n can mean a loss
// in n+1 ply and a return value +n can mean a win in n+1 ply. This
// cannot happen for tables with positions exactly on the "edge" of
// the 50-move rule.
//
// This implies that if dtz > 0 is returned, the position is certainly
// a win if dtz + 50-move-counter <= 99. Care must be taken that the engine
// picks moves that preserve dtz + 50-move-counter <= 99.
//
// If n = 100 immediately after a capture or pawn move, then the position
// is also certainly a win, and during the whole phase until the next
// capture or pawn move, the inequality to be preserved is
// dtz + 50-move-counter <= 100.
//
// In short, if a move is available resulting in dtz + 50-move-counter <= 99,
// then do not accept moves leading to dtz + 50-move-counter == 100.
int Tablebases::probe_dtz(Position& pos, ProbeState* result) {

    *result = OK;
    WDLScore wdl = search<true>(pos, result);

    if (*result == FAIL || wdl == WDLDraw) // DTZ tables don't store draws
        return 0;

    // DTZ stores a 'don't care value in this case, or even a plain wrong
    // one as in case the best move is a losing ep, so it cannot be probed.
    if (*result == ZEROING_BEST_MOVE)
        return dtz_before_zeroing(wdl);

    int dtz = probe_table<DTZ>(pos, result, wdl);

    if (*result == FAIL)
        return 0;

    if (*result != CHANGE_STM)
        return (dtz + 100 * (wdl == WDLBlessedLoss || wdl == WDLCursedWin)) * sign_of(wdl);

    // DTZ stores results for the other side, so we need to do a 1-ply search and
    // find the winning move that minimizes DTZ.
    StateInfo st;
    int       minDTZ = 0xFFFF;

    for (const Move move : MoveList<LEGAL>(pos))
    {
        bool zeroing = pos.capture(move) || type_of(pos.moved_piece(move)) == PAWN;

        pos.do_move(move, st);

        // For zeroing moves we want the dtz of the move _before_ doing it,
        // otherwise we will get the dtz of the next move sequence. Search the
        // position after the move to get the score sign (because even in a
        // winning position we could make a losing capture or go for a draw).
        dtz = zeroing ? -dtz_before_zeroing(search<false>(pos, result)) : -probe_dtz(pos, result);

        // If the move mates, force minDTZ to 1
        if (dtz == 1 && pos.checkers() && MoveList<LEGAL>(pos).size() == 0)
            minDTZ = 1;

        // Convert result from 1-ply search. Zeroing moves are already accounted
        // by dtz_before_zeroing() that returns the DTZ of the previous move.
        if (!zeroing)
            dtz += sign_of(dtz);

        // Skip the draws and if we are winning only pick positive dtz
        if (dtz < minDTZ && sign_of(dtz) == sign_of(wdl))
            minDTZ = dtz;

        pos.undo_move(move);

        if (*result == FAIL)
            return 0;
    }

    // When there are no legal moves, the position is mate: we return -1
    return minDTZ == 0xFFFF ? -1 : minDTZ;
}

// Use the DTZ tables to rank root moves.
//
// A return value false indicates that not all probes were successful.
bool Tablebases::root_probe(Position&          pos,
                            Search::RootMoves& rootMoves,
                            bool               rule50,
                            bool               rankDTZ) {

    ProbeState result = OK;
    StateInfo  st;

    // Obtain 50-move counter for the root position
    int cnt50 = pos.rule50_count();

    // Check whether a position was repeated since the last zeroing move.
    bool rep = pos.has_repeated();

    int dtz, bound = rule50 ? (MAX_DTZ / 2 - 100) : 1;

    // Probe and rank each move
    for (auto& m : rootMoves)
    {
        pos.do_move(m.pv[0], st);

        // Calculate dtz for the current move counting from the root position
        if (pos.rule50_count() == 0)
        {
            // In case of a zeroing move, dtz is one of -101/-1/0/1/101
            WDLScore wdl = -probe_wdl(pos, &result);
            dtz          = dtz_before_zeroing(wdl);
        }
        else if (pos.is_draw(1))
        {
            // In case a root move leads to a draw by repetition or 50-move rule,
            // we set dtz to zero. Note: since we are only 1 ply from the root,
            // this must be a true 3-fold repetition inside the game history.
            dtz = 0;
        }
        else
        {
            // Otherwise, take dtz for the new position and correct by 1 ply
            dtz = -probe_dtz(pos, &result);
            dtz = dtz > 0 ? dtz + 1 : dtz < 0 ? dtz - 1 : dtz;
        }

        // Make sure that a mating move is assigned a dtz value of 1
        if (pos.checkers() && dtz == 2 && MoveList<LEGAL>(pos).size() == 0)
            dtz = 1;

        pos.undo_move(m.pv[0]);

        if (result == FAIL)
            return false;

        // Better moves are ranked higher. Certain wins are ranked equally.
        // Losing moves are ranked equally unless a 50-move draw is in sight.
        int r     = dtz > 0 ? (dtz + cnt50 <= 99 && !rep ? MAX_DTZ - (rankDTZ ? dtz : 0)
                                                         : MAX_DTZ / 2 - (dtz + cnt50))
                  : dtz < 0 ? (-dtz * 2 + cnt50 < 100 ? -MAX_DTZ - (rankDTZ ? dtz : 0)
                                                      : -MAX_DTZ / 2 + (-dtz + cnt50))
                            : 0;
        m.tbRank = r;

        // Determine the score to be displayed for this move. Assign at least
        // 1 cp to cursed wins and let it grow to 49 cp as the positions gets
        // closer to a real win.
        m.tbScore = r >= bound ? VALUE_MATE - MAX_PLY - 1
                  : r > 0      ? Value((std::max(3, r - (MAX_DTZ / 2 - 200)) * int(PawnValue)) / 200)
                  : r == 0     ? VALUE_DRAW
                  : r > -bound
                    ? Value((std::min(-3, r + (MAX_DTZ / 2 - 200)) * int(PawnValue)) / 200)
                    : -VALUE_MATE + MAX_PLY + 1;
    }

    return true;
}

// Use the WDL tables to rank root moves.
// This is a fallback for the case that some or all DTZ tables are missing.
//
// A return value false indicates that not all probes were successful.
bool Tablebases::root_probe_wdl(Position& pos, Search::RootMoves& rootMoves, bool rule50) {

    static const int WDL_to_rank[] = {-MAX_DTZ, -MAX_DTZ + 101, 0, MAX_DTZ - 101, MAX_DTZ};

    ProbeState result = OK;
    StateInfo  st;
    WDLScore   wdl;

    // Probe and rank each move
    for (auto& m : rootMoves)
    {
        pos.do_move(m.pv[0], st);

        if (pos.is_draw(1))
            wdl = WDLDraw;
        else
            wdl = -probe_wdl(pos, &result);

        pos.undo_move(m.pv[0]);

        if (result == FAIL)
            return false;

        m.tbRank = WDL_to_rank[wdl + 2];

        if (!rule50)
            wdl = wdl > WDLDraw ? WDLWin : wdl < WDLDraw ? WDLLoss : WDLDraw;
        m.tbScore = WDL_to_value[wdl + 2];
    }

    return true;
}

Config Tablebases::rank_root_moves(const OptionsMap&  options,
                                   Position&          pos,
                                   Search::RootMoves& rootMoves,
                                   bool               rankDTZ) {
    Config config;

    if (rootMoves.empty())
        return config;

    config.rootInTB    = false;
    config.useRule50   = bool(options["Syzygy50MoveRule"]);
    config.probeDepth  = int(options["SyzygyProbeDepth"]);
    config.cardinality = int(options["SyzygyProbeLimit"]);

    bool dtz_available = true;

    // Tables with fewer pieces than SyzygyProbeLimit are searched with
    // probeDepth == DEPTH_ZERO
    if (config.cardinality > MaxCardinality)
    {
        config.cardinality = MaxCardinality;
        config.probeDepth  = 0;
    }

    if (config.cardinality >= popcount(pos.pieces()) && !pos.can_castle(ANY_CASTLING))
    {
        // Rank moves using DTZ tables
        config.rootInTB = root_probe(pos, rootMoves, options["Syzygy50MoveRule"], rankDTZ);

        if (!config.rootInTB)
        {
            // DTZ tables are missing; try to rank moves using WDL tables
            dtz_available   = false;
            config.rootInTB = root_probe_wdl(pos, rootMoves, options["Syzygy50MoveRule"]);
        }
    }

    if (config.rootInTB)
    {
        // Sort moves according to TB rank
        std::stable_sort(
          rootMoves.begin(), rootMoves.end(),
          [](const Search::RootMove& a, const Search::RootMove& b) { return a.tbRank > b.tbRank; });

        // Probe during search only if DTZ is not available and we are winning
        if (dtz_available || rootMoves[0].tbScore <= VALUE_DRAW)
            config.cardinality = 0;
    }
    else
    {
        // Clean up if root_probe() and root_probe_wdl() have failed
        for (auto& m : rootMoves)
            m.tbRank = 0;
    }

    return config;
}
}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

using std::string;

namespace Stockfish {

bool          Tune::update_on_last;
const Option* LastOption = nullptr;
OptionsMap*   Tune::options;
namespace {
std::map<std::string, int> TuneResults;

std::optional<std::string> on_tune(const Option& o) {

    if (!Tune::update_on_last || LastOption == &o)
        Tune::read_options();

    return std::nullopt;
}
}

void Tune::make_option(OptionsMap* opts, const string& n, int v, const SetRange& r) {

    // Do not generate option when there is nothing to tune (ie. min = max)
    if (r(v).first == r(v).second)
        return;

    if (TuneResults.count(n))
        v = TuneResults[n];

    (*opts)[n] << Option(v, r(v).first, r(v).second, on_tune);
    LastOption = &((*opts)[n]);

    // Print formatted parameters, ready to be copy-pasted in Fishtest
    std::cout << n << ","                                  //
              << v << ","                                  //
              << r(v).first << ","                         //
              << r(v).second << ","                        //
              << (r(v).second - r(v).first) / 20.0 << ","  //
              << "0.0020" << std::endl;
}

string Tune::next(string& names, bool pop) {

    string name;

    do
    {
        string token = names.substr(0, names.find(','));

        if (pop)
            names.erase(0, token.size() + 1);

        std::stringstream ws(token);
        name += (ws >> token, token);  // Remove trailing whitespace

    } while (std::count(name.begin(), name.end(), '(') - std::count(name.begin(), name.end(), ')'));

    return name;
}

template<>
void Tune::Entry<int>::init_option() {
    make_option(options, name, value, range);
}

template<>
void Tune::Entry<int>::read_option() {
    if (options->count(name))
        value = int((*options)[name]);
}

// Instead of a variable here we have a PostUpdate function: just call it
template<>
void Tune::Entry<Tune::PostUpdate>::init_option() {}
template<>
void Tune::Entry<Tune::PostUpdate>::read_option() {
    value();
}

}  // namespace Stockfish

// Init options with tuning session results instead of default values. Useful to
// get correct bench signature after a tuning session or to test tuned values.
// Just copy fishtest tuning results in a result.txt file and extract the
// values with:
//
// cat results.txt | sed 's/^param: \([^,]*\), best: \([^,]*\).*/  TuneResults["\1"] = int(round(\2));/'
//
// Then paste the output below, as the function body

namespace Stockfish {

void Tune::read_results() { /* ...insert your values here... */ }

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

//Definition of input features HalfKAv2_hm of NNUE evaluation function

namespace Stockfish::Eval::NNUE::Features {

// Index of a feature for a given king position and another piece on some square
template<Color Perspective>
inline IndexType HalfKAv2_hm::make_index(Square s, Piece pc, Square ksq) {
    return IndexType((int(s) ^ OrientTBL[Perspective][ksq]) + PieceSquareIndex[Perspective][pc]
                     + KingBuckets[Perspective][ksq]);
}

// Get a list of indices for active features
template<Color Perspective>
void HalfKAv2_hm::append_active_indices(const Position& pos, IndexList& active) {
    Square   ksq = pos.square<KING>(Perspective);
    Bitboard bb  = pos.pieces();
    while (bb)
    {
        Square s = pop_lsb(bb);
        active.push_back(make_index<Perspective>(s, pos.piece_on(s), ksq));
    }
}

// Explicit template instantiations
template void HalfKAv2_hm::append_active_indices<WHITE>(const Position& pos, IndexList& active);
template void HalfKAv2_hm::append_active_indices<BLACK>(const Position& pos, IndexList& active);
template IndexType HalfKAv2_hm::make_index<WHITE>(Square s, Piece pc, Square ksq);
template IndexType HalfKAv2_hm::make_index<BLACK>(Square s, Piece pc, Square ksq);

// Get a list of indices for recently changed features
template<Color Perspective>
void HalfKAv2_hm::append_changed_indices(Square            ksq,
                                         const DirtyPiece& dp,
                                         IndexList&        removed,
                                         IndexList&        added) {
    for (int i = 0; i < dp.dirty_num; ++i)
    {
        if (dp.from[i] != SQ_NONE)
            removed.push_back(make_index<Perspective>(dp.from[i], dp.piece[i], ksq));
        if (dp.to[i] != SQ_NONE)
            added.push_back(make_index<Perspective>(dp.to[i], dp.piece[i], ksq));
    }
}

// Explicit template instantiations
template void HalfKAv2_hm::append_changed_indices<WHITE>(Square            ksq,
                                                         const DirtyPiece& dp,
                                                         IndexList&        removed,
                                                         IndexList&        added);
template void HalfKAv2_hm::append_changed_indices<BLACK>(Square            ksq,
                                                         const DirtyPiece& dp,
                                                         IndexList&        removed,
                                                         IndexList&        added);

int HalfKAv2_hm::update_cost(const StateInfo* st) { return st->dirtyPiece.dirty_num; }

int HalfKAv2_hm::refresh_cost(const Position& pos) { return pos.count<ALL_PIECES>(); }

bool HalfKAv2_hm::requires_refresh(const StateInfo* st, Color perspective) {
    return st->dirtyPiece.piece[0] == make_piece(perspective, KING);
}

}  // namespace Stockfish::Eval::NNUE::Features

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

using std::string;

namespace Stockfish {

namespace Zobrist {

Key psq[PIECE_NB][SQUARE_NB];
Key enpassant[FILE_NB];
Key castling[CASTLING_RIGHT_NB];
Key side, noPawns;
}

namespace {

constexpr Piece Pieces[] = {W_PAWN, W_KNIGHT, W_BISHOP, W_ROOK, W_QUEEN, W_KING,
                            B_PAWN, B_KNIGHT, B_BISHOP, B_ROOK, B_QUEEN, B_KING};
}  // namespace

// Returns an ASCII representation of the position
std::ostream& operator<<(std::ostream& os, const Position& pos) {

    os << "\n +---+---+---+---+---+---+---+---+\n";

    for (Rank r = RANK_8; r >= RANK_1; --r)
    {
        for (File f = FILE_A; f <= FILE_H; ++f)
            os << " | " << PieceToChar[pos.piece_on(make_square(f, r))];

        os << " | " << (1 + r) << "\n +---+---+---+---+---+---+---+---+\n";
    }

    os << "   a   b   c   d   e   f   g   h\n"
       << "\nFen: " << pos.fen() << "\nKey: " << std::hex << std::uppercase << std::setfill('0')
       << std::setw(16) << pos.key() << std::setfill(' ') << std::dec << "\nCheckers: ";

    for (Bitboard b = pos.checkers(); b;)
        os << UCIEngine::square(pop_lsb(b)) << " ";

    if (int(Tablebases::MaxCardinality) >= popcount(pos.pieces()) && !pos.can_castle(ANY_CASTLING))
    {
        StateInfo st;
        ASSERT_ALIGNED(&st, Eval::NNUE::CacheLineSize);

        Position p;
        p.set(pos.fen(), pos.is_chess960(), &st);
        Tablebases::ProbeState s1, s2;
        Tablebases::WDLScore   wdl = Tablebases::probe_wdl(p, &s1);
        int                    dtz = Tablebases::probe_dtz(p, &s2);
        os << "\nTablebases WDL: " << std::setw(4) << wdl << " (" << s1 << ")"
           << "\nTablebases DTZ: " << std::setw(4) << dtz << " (" << s2 << ")";
    }

    return os;
}

// Implements Marcel van Kervinck's cuckoo algorithm to detect repetition of positions
// for 3-fold repetition draws. The algorithm uses two hash tables with Zobrist hashes
// to allow fast detection of recurring positions. For details see:
// http://web.archive.org/web/20201107002606/https://marcelk.net/2013-04-06/paper/upcoming-rep-v2.pdf

// First and second hash functions for indexing the cuckoo tables
inline int H1(Key h) { return h & 0x1fff; }
inline int H2(Key h) { return (h >> 16) & 0x1fff; }

// Cuckoo tables with Zobrist hashes of valid reversible moves, and the moves themselves
std::array<Key, 8192>  cuckoo;
std::array<Move, 8192> cuckooMove;

// Initializes at startup the various arrays used to compute hash keys
void Position::init() {

    PRNG rng(1070372);

    for (Piece pc : Pieces)
        for (Square s = SQ_A1; s <= SQ_H8; ++s)
            Zobrist::psq[pc][s] = rng.rand<Key>();

    for (File f = FILE_A; f <= FILE_H; ++f)
        Zobrist::enpassant[f] = rng.rand<Key>();

    for (int cr = NO_CASTLING; cr <= ANY_CASTLING; ++cr)
        Zobrist::castling[cr] = rng.rand<Key>();

    Zobrist::side    = rng.rand<Key>();
    Zobrist::noPawns = rng.rand<Key>();

    // Prepare the cuckoo tables
    cuckoo.fill(0);
    cuckooMove.fill(Move::none());
    [[maybe_unused]] int count = 0;
    for (Piece pc : Pieces)
        for (Square s1 = SQ_A1; s1 <= SQ_H8; ++s1)
            for (Square s2 = Square(s1 + 1); s2 <= SQ_H8; ++s2)
                if ((type_of(pc) != PAWN) && (attacks_bb(type_of(pc), s1, 0) & s2))
                {
                    Move move = Move(s1, s2);
                    Key  key  = Zobrist::psq[pc][s1] ^ Zobrist::psq[pc][s2] ^ Zobrist::side;
                    int  i    = H1(key);
                    while (true)
                    {
                        std::swap(cuckoo[i], key);
                        std::swap(cuckooMove[i], move);
                        if (move == Move::none())  // Arrived at empty slot?
                            break;
                        i = (i == H1(key)) ? H2(key) : H1(key);  // Push victim to alternative slot
                    }
                    count++;
                }
    assert(count == 3668);
}

// Initializes the position object with the given FEN string.
// This function is not very robust - make sure that input FENs are correct,
// this is assumed to be the responsibility of the GUI.
Position& Position::set(const string& fenStr, bool isChess960, StateInfo* si) {
    /*
   A FEN string defines a particular position using only the ASCII character set.

   A FEN string contains six fields separated by a space. The fields are:

   1) Piece placement (from white's perspective). Each rank is described, starting
      with rank 8 and ending with rank 1. Within each rank, the contents of each
      square are described from file A through file H. Following the Standard
      Algebraic Notation (SAN), each piece is identified by a single letter taken
      from the standard English names. White pieces are designated using upper-case
      letters ("PNBRQK") whilst Black uses lowercase ("pnbrqk"). Blank squares are
      noted using digits 1 through 8 (the number of blank squares), and "/"
      separates ranks.

   2) Active color. "w" means white moves next, "b" means black.

   3) Castling availability. If neither side can castle, this is "-". Otherwise,
      this has one or more letters: "K" (White can castle kingside), "Q" (White
      can castle queenside), "k" (Black can castle kingside), and/or "q" (Black
      can castle queenside).

   4) En passant target square (in algebraic notation). If there's no en passant
      target square, this is "-". If a pawn has just made a 2-square move, this
      is the position "behind" the pawn. Following X-FEN standard, this is recorded
      only if there is a pawn in position to make an en passant capture, and if
      there really is a pawn that might have advanced two squares.

   5) Halfmove clock. This is the number of halfmoves since the last pawn advance
      or capture. This is used to determine if a draw can be claimed under the
      fifty-move rule.

   6) Fullmove number. The number of the full move. It starts at 1, and is
      incremented after Black's move.
*/

    unsigned char      col, row, token;
    size_t             idx;
    Square             sq = SQ_A8;
    std::istringstream ss(fenStr);

    std::memset(this, 0, sizeof(Position));
    std::memset(si, 0, sizeof(StateInfo));
    st = si;

    ss >> std::noskipws;

    // 1. Piece placement
    while ((ss >> token) && !isspace(token))
    {
        if (isdigit(token))
            sq += (token - '0') * EAST;  // Advance the given number of files

        else if (token == '/')
            sq += 2 * SOUTH;

        else if ((idx = PieceToChar.find(token)) != string::npos)
        {
            put_piece(Piece(idx), sq);
            ++sq;
        }
    }

    // 2. Active color
    ss >> token;
    sideToMove = (token == 'w' ? WHITE : BLACK);
    ss >> token;

    // 3. Castling availability. Compatible with 3 standards: Normal FEN standard,
    // Shredder-FEN that uses the letters of the columns on which the rooks began
    // the game instead of KQkq and also X-FEN standard that, in case of Chess960,
    // if an inner rook is associated with the castling right, the castling tag is
    // replaced by the file letter of the involved rook, as for the Shredder-FEN.
    while ((ss >> token) && !isspace(token))
    {
        Square rsq;
        Color  c    = islower(token) ? BLACK : WHITE;
        Piece  rook = make_piece(c, ROOK);

        token = char(toupper(token));

        if (token == 'K')
            for (rsq = relative_square(c, SQ_H1); piece_on(rsq) != rook; --rsq)
            {}

        else if (token == 'Q')
            for (rsq = relative_square(c, SQ_A1); piece_on(rsq) != rook; ++rsq)
            {}

        else if (token >= 'A' && token <= 'H')
            rsq = make_square(File(token - 'A'), relative_rank(c, RANK_1));

        else
            continue;

        set_castling_right(c, rsq);
    }

    // 4. En passant square.
    // Ignore if square is invalid or not on side to move relative rank 6.
    bool enpassant = false;

    if (((ss >> col) && (col >= 'a' && col <= 'h'))
        && ((ss >> row) && (row == (sideToMove == WHITE ? '6' : '3'))))
    {
        st->epSquare = make_square(File(col - 'a'), Rank(row - '1'));

        // En passant square will be considered only if
        // a) side to move have a pawn threatening epSquare
        // b) there is an enemy pawn in front of epSquare
        // c) there is no piece on epSquare or behind epSquare
        enpassant = pawn_attacks_bb(~sideToMove, st->epSquare) & pieces(sideToMove, PAWN)
                 && (pieces(~sideToMove, PAWN) & (st->epSquare + pawn_push(~sideToMove)))
                 && !(pieces() & (st->epSquare | (st->epSquare + pawn_push(sideToMove))));
    }

    if (!enpassant)
        st->epSquare = SQ_NONE;

    // 5-6. Halfmove clock and fullmove number
    ss >> std::skipws >> st->rule50 >> gamePly;

    // Convert from fullmove starting from 1 to gamePly starting from 0,
    // handle also common incorrect FEN with fullmove = 0.
    gamePly = std::max(2 * (gamePly - 1), 0) + (sideToMove == BLACK);

    chess960 = isChess960;
    set_state();

    assert(pos_is_ok());

    return *this;
}

// Helper function used to set castling
// rights given the corresponding color and the rook starting square.
void Position::set_castling_right(Color c, Square rfrom) {

    Square         kfrom = square<KING>(c);
    CastlingRights cr    = c & (kfrom < rfrom ? KING_SIDE : QUEEN_SIDE);

    st->castlingRights |= cr;
    castlingRightsMask[kfrom] |= cr;
    castlingRightsMask[rfrom] |= cr;
    castlingRookSquare[cr] = rfrom;

    Square kto = relative_square(c, cr & KING_SIDE ? SQ_G1 : SQ_C1);
    Square rto = relative_square(c, cr & KING_SIDE ? SQ_F1 : SQ_D1);

    castlingPath[cr] = (between_bb(rfrom, rto) | between_bb(kfrom, kto)) & ~(kfrom | rfrom);
}

// Sets king attacks to detect if a move gives check
void Position::set_check_info() const {

    update_slider_blockers(WHITE);
    update_slider_blockers(BLACK);

    Square ksq = square<KING>(~sideToMove);

    st->checkSquares[PAWN]   = pawn_attacks_bb(~sideToMove, ksq);
    st->checkSquares[KNIGHT] = attacks_bb<KNIGHT>(ksq);
    st->checkSquares[BISHOP] = attacks_bb<BISHOP>(ksq, pieces());
    st->checkSquares[ROOK]   = attacks_bb<ROOK>(ksq, pieces());
    st->checkSquares[QUEEN]  = st->checkSquares[BISHOP] | st->checkSquares[ROOK];
    st->checkSquares[KING]   = 0;
}

// Computes the hash keys of the position, and other
// data that once computed is updated incrementally as moves are made.
// The function is only used when a new position is set up
void Position::set_state() const {

    st->key = st->materialKey = 0;
    st->pawnKey               = Zobrist::noPawns;
    st->nonPawnMaterial[WHITE] = st->nonPawnMaterial[BLACK] = VALUE_ZERO;
    st->checkersBB             = attackers_to(square<KING>(sideToMove)) & pieces(~sideToMove);

    set_check_info();

    for (Bitboard b = pieces(); b;)
    {
        Square s  = pop_lsb(b);
        Piece  pc = piece_on(s);
        st->key ^= Zobrist::psq[pc][s];

        if (type_of(pc) == PAWN)
            st->pawnKey ^= Zobrist::psq[pc][s];

        else if (type_of(pc) != KING)
            st->nonPawnMaterial[color_of(pc)] += PieceValue[pc];
    }

    if (st->epSquare != SQ_NONE)
        st->key ^= Zobrist::enpassant[file_of(st->epSquare)];

    if (sideToMove == BLACK)
        st->key ^= Zobrist::side;

    st->key ^= Zobrist::castling[st->castlingRights];

    for (Piece pc : Pieces)
        for (int cnt = 0; cnt < pieceCount[pc]; ++cnt)
            st->materialKey ^= Zobrist::psq[pc][cnt];
}

// Overload to initialize the position object with the given endgame code string
// like "KBPKN". It's mainly a helper to get the material key out of an endgame code.
Position& Position::set(const string& code, Color c, StateInfo* si) {

    assert(code[0] == 'K');

    string sides[] = {code.substr(code.find('K', 1)),                                 // Weak
                      code.substr(0, std::min(code.find('v'), code.find('K', 1)))};  // Strong

    assert(sides[0].length() > 0 && sides[0].length() < 8);
    assert(sides[1].length() > 0 && sides[1].length() < 8);

    std::transform(sides[c].begin(), sides[c].end(), sides[c].begin(), tolower);

    string fenStr = "8/" + sides[0] + char(8 - sides[0].length() + '0') + "/8/8/8/8/" + sides[1]
                  + char(8 - sides[1].length() + '0') + "/8 w - - 0 10";

    return set(fenStr, false, si);
}

// Returns a FEN representation of the position. In case of
// Chess960 the Shredder-FEN notation is used. This is mainly a debugging function.
string Position::fen() const {

    int                emptyCnt;
    std::ostringstream ss;

    for (Rank r = RANK_8; r >= RANK_1; --r)
    {
        for (File f = FILE_A; f <= FILE_H; ++f)
        {
            for (emptyCnt = 0; f <= FILE_H && empty(make_square(f, r)); ++f)
                ++emptyCnt;

            if (emptyCnt)
                ss << emptyCnt;

            if (f <= FILE_H)
                ss << PieceToChar[piece_on(make_square(f, r))];
        }

        if (r > RANK_1)
            ss << '/';
    }

    ss << (sideToMove == WHITE ? " w " : " b ");

    if (can_castle(WHITE_OO))
        ss << (chess960 ? char('A' + file_of(castling_rook_square(WHITE_OO))) : 'K');

    if (can_castle(WHITE_OOO))
        ss << (chess960 ? char('A' + file_of(castling_rook_square(WHITE_OOO))) : 'Q');

    if (can_castle(BLACK_OO))
        ss << (chess960 ? char('a' + file_of(castling_rook_square(BLACK_OO))) : 'k');

    if (can_castle(BLACK_OOO))
        ss << (chess960 ? char('a' + file_of(castling_rook_square(BLACK_OOO))) : 'q');

    if (!can_castle(ANY_CASTLING))
        ss << '-';

    ss << (ep_square() == SQ_NONE ? " - " : " " + UCIEngine::square(ep_square()) + " ")
       << st->rule50 << " " << 1 + (gamePly - (sideToMove == BLACK)) / 2;

    return ss.str();
}

// Calculates st->blockersForKing[c] and st->pinners[~c],
// which store respectively the pieces preventing king of color c from being in check
// and the slider pieces of color ~c pinning pieces of color c to the king.
void Position::update_slider_blockers(Color c) const {

    Square ksq = square<KING>(c);

    st->blockersForKing[c] = 0;
    st->pinners[~c]        = 0;

    // Snipers are sliders that attack 's' when a piece and other snipers are removed
    Bitboard snipers = ((attacks_bb<ROOK>(ksq) & pieces(QUEEN, ROOK))
                        | (attacks_bb<BISHOP>(ksq) & pieces(QUEEN, BISHOP)))
                     & pieces(~c);
    Bitboard occupancy = pieces() ^ snipers;

    while (snipers)
    {
        Square   sniperSq = pop_lsb(snipers);
        Bitboard b        = between_bb(ksq, sniperSq) & occupancy;

        if (b && !more_than_one(b))
        {
            st->blockersForKing[c] |= b;
            if (b & pieces(c))
                st->pinners[~c] |= sniperSq;
        }
    }
}

// Computes a bitboard of all pieces which attack a given square.
// Slider attacks use the occupied bitboard to indicate occupancy.
Bitboard Position::attackers_to(Square s, Bitboard occupied) const {

    return (pawn_attacks_bb(BLACK, s) & pieces(WHITE, PAWN))
         | (pawn_attacks_bb(WHITE, s) & pieces(BLACK, PAWN))
         | (attacks_bb<KNIGHT>(s) & pieces(KNIGHT))
         | (attacks_bb<ROOK>(s, occupied) & pieces(ROOK, QUEEN))
         | (attacks_bb<BISHOP>(s, occupied) & pieces(BISHOP, QUEEN))
         | (attacks_bb<KING>(s) & pieces(KING));
}

// Tests whether a pseudo-legal move is legal
bool Position::legal(Move m) const {

    assert(m.is_ok());

    Color  us   = sideToMove;
    Square from = m.from_sq();
    Square to   = m.to_sq();

    assert(color_of(moved_piece(m)) == us);
    assert(piece_on(square<KING>(us)) == make_piece(us, KING));

    // En passant captures are a tricky special case. Because they are rather
    // uncommon, we do it simply by testing whether the king is attacked after
    // the move is made.
    if (m.type_of() == EN_PASSANT)
    {
        Square   ksq      = square<KING>(us);
        Square   capsq    = to - pawn_push(us);
        Bitboard occupied = (pieces() ^ from ^ capsq) | to;

        assert(to == ep_square());
        assert(moved_piece(m) == make_piece(us, PAWN));
        assert(piece_on(capsq) == make_piece(~us, PAWN));
        assert(piece_on(to) == NO_PIECE);

        return !(attacks_bb<ROOK>(ksq, occupied) & pieces(~us, QUEEN, ROOK))
            && !(attacks_bb<BISHOP>(ksq, occupied) & pieces(~us, QUEEN, BISHOP));
    }

    // Castling moves generation does not check if the castling path is clear of
    // enemy attacks, it is delayed at a later time: now!
    if (m.type_of() == CASTLING)
    {
        // After castling, the rook and king final positions are the same in
        // Chess960 as they would be in standard chess.
        to             = relative_square(us, to > from ? SQ_G1 : SQ_C1);
        Direction step = to > from ? WEST : EAST;

        for (Square s = to; s != from; s += step)
            if (attackers_to(s) & pieces(~us))
                return false;

        // In case of Chess960, verify if the Rook blocks some checks.
        // For instance an enemy queen in SQ_A1 when castling rook is in SQ_B1.
        return !chess960 || !(blockers_for_king(us) & m.to_sq());
    }

    // If the moving piece is a king, check whether the destination square is
    // attacked by the opponent.
    if (type_of(piece_on(from)) == KING)
        return !(attackers_to(to, pieces() ^ from) & pieces(~us));

    // A non-king move is legal if and only if it is not pinned or it
    // is moving along the ray towards or away from the king.
    return !(blockers_for_king(us) & from) || aligned(from, to, square<KING>(us));
}

// Takes a random move and tests whether the move is
// pseudo-legal. It is used to validate moves from TT that can be corrupted
// due to SMP concurrent access or hash position key aliasing.
bool Position::pseudo_legal(const Move m) const {

    Color  us   = sideToMove;
    Square from = m.from_sq();
    Square to   = m.to_sq();
    Piece  pc   = moved_piece(m);
```

```cpp
    // Use a slower but simpler function for uncommon cases
    // yet we skip the legality check of MoveList<LEGAL>().
    if (m.type_of() != NORMAL)
        return checkers() ? MoveList<EVASIONS>(*this).contains(m)
                          : MoveList<NON_EVASIONS>(*this).contains(m);

    // Is not a promotion, so the promotion piece must be empty
    assert(m.promotion_type() - KNIGHT == NO_PIECE_TYPE);

    // If the 'from' square is not occupied by a piece belonging to the side to
    // move, the move is obviously not legal.
    if (pc == NO_PIECE || color_of(pc) != us)
        return false;

    // The destination square cannot be occupied by a friendly piece
    if (pieces(us) & to)
        return false;

    // Handle the special case of a pawn move
    if (type_of(pc) == PAWN)
    {
        // We have already handled promotion moves, so destination cannot be on the 8th/1st rank
        if ((Rank8BB | Rank1BB) & to)
            return false;

        if (!(pawn_attacks_bb(us, from) & pieces(~us) & to)  // Not a capture
            && !((from + pawn_push(us) == to) && empty(to))  // Not a single push
            && !((from + 2 * pawn_push(us) == to)             // Not a double push
                 && (relative_rank(us, from) == RANK_2) && empty(to) && empty(to - pawn_push(us))))
            return false;
    }
    else if (!(attacks_bb(type_of(pc), from, pieces()) & to))
        return false;

    // Evasions generator already takes care to avoid some kind of illegal moves
    // and legal() relies on this. We therefore have to take care that the same
    // kind of moves are filtered out here.
    if (checkers())
    {
        if (type_of(pc) != KING)
        {
            // Double check? In this case, a king move is required
            if (more_than_one(checkers()))
                return false;

            // Our move must be a blocking interposition or a capture of the checking piece
            if (!(between_bb(square<KING>(us), lsb(checkers())) & to))
                return false;
        }
        // In case of king moves under check we have to remove the king so as to catch
        // invalid moves like b1a1 when opposite queen is on c1.
        else if (attackers_to(to, pieces() ^ from) & pieces(~us))
            return false;
    }

    return true;
}

// Tests whether a pseudo-legal move gives a check
bool Position::gives_check(Move m) const {

    assert(m.is_ok());
    assert(color_of(moved_piece(m)) == sideToMove);

    Square from = m.from_sq();
    Square to   = m.to_sq();

    // Is there a direct check?
    if (check_squares(type_of(piece_on(from))) & to)
        return true;

    // Is there a discovered check?
    if (blockers_for_king(~sideToMove) & from)
        return !aligned(from, to, square<KING>(~sideToMove)) || m.type_of() == CASTLING;

    switch (m.type_of())
    {
    case NORMAL :
        return false;

    case PROMOTION :
        return attacks_bb(m.promotion_type(), to, pieces() ^ from) & square<KING>(~sideToMove);

    // En passant capture with check? We have already handled the case of direct
    // checks and ordinary discovered check, so the only case we need to handle
    // is the unusual case of a discovered check through the captured pawn.
    case EN_PASSANT : {
        Square   capsq = make_square(file_of(to), rank_of(from));
        Bitboard b     = (pieces() ^ from ^ capsq) | to;

        return (attacks_bb<ROOK>(square<KING>(~sideToMove), b) & pieces(sideToMove, QUEEN, ROOK))
             | (attacks_bb<BISHOP>(square<KING>(~sideToMove), b)
                & pieces(sideToMove, QUEEN, BISHOP));
    }
    default :  //CASTLING
    {
        // Castling is encoded as 'king captures the rook'
        Square rto = relative_square(sideToMove, to > from ? SQ_F1 : SQ_D1);

        return check_squares(ROOK) & rto;
    }
    }
}

// Makes a move, and saves all information necessary
// to a StateInfo object. The move is assumed to be legal. Pseudo-legal
// moves should be filtered out before this function is called.
void Position::do_move(Move m, StateInfo& newSt, bool givesCheck) {

    assert(m.is_ok());
    assert(&newSt != st);

    Key k = st->key ^ Zobrist::side;

    // Copy some fields of the old state to our new StateInfo object except the
    // ones which are going to be recalculated from scratch anyway and then switch
    // our state pointer to point to the new (ready to be updated) state.
    std::memcpy(&newSt, st, offsetof(StateInfo, key));
    newSt.previous = st;
    st             = &newSt;

    // Increment ply counters. In particular, rule50 will be reset to zero later on
    // in case of a capture or a pawn move.
    ++gamePly;
    ++st->rule50;
    ++st->pliesFromNull;

    // Used by NNUE
    st->accumulatorBig.computed[WHITE]     = st->accumulatorBig.computed[BLACK] =
      st->accumulatorSmall.computed[WHITE] = st->accumulatorSmall.computed[BLACK] = false;

    auto& dp     = st->dirtyPiece;
    dp.dirty_num = 1;

    Color  us       = sideToMove;
    Color  them     = ~us;
    Square from     = m.from_sq();
    Square to       = m.to_sq();
    Piece  pc       = piece_on(from);
    Piece  captured = m.type_of() == EN_PASSANT ? make_piece(them, PAWN) : piece_on(to);

    assert(color_of(pc) == us);
    assert(captured == NO_PIECE || color_of(captured) == (m.type_of() != CASTLING ? them : us));
    assert(type_of(captured) != KING);

    if (m.type_of() == CASTLING)
    {
        assert(pc == make_piece(us, KING));
        assert(captured == make_piece(us, ROOK));

        Square rfrom, rto;
        do_castling<true>(us, from, to, rfrom, rto);

        k ^= Zobrist::psq[captured][rfrom] ^ Zobrist::psq[captured][rto];
        captured = NO_PIECE;
    }

    if (captured)
    {
        Square capsq = to;

        // If the captured piece is a pawn, update pawn hash key, otherwise
        // update non-pawn material.
        if (type_of(captured) == PAWN)
        {
            if (m.type_of() == EN_PASSANT)
            {
                capsq -= pawn_push(us);

                assert(pc == make_piece(us, PAWN));
                assert(to == st->epSquare);
                assert(relative_rank(us, to) == RANK_6);
                assert(piece_on(to) == NO_PIECE);
                assert(piece_on(capsq) == make_piece(them, PAWN));
            }

            st->pawnKey ^= Zobrist::psq[captured][capsq];
        }
        else
            st->nonPawnMaterial[them] -= PieceValue[captured];

        dp.dirty_num = 2;  // 1 piece moved, 1 piece captured
        dp.piece[1]  = captured;
        dp.from[1]   = capsq;
        dp.to[1]     = SQ_NONE;

        // Update board and piece lists
        remove_piece(capsq);

        k ^= Zobrist::psq[captured][capsq];
        st->materialKey ^= Zobrist::psq[captured][pieceCount[captured]];

        // Reset rule 50 counter
        st->rule50 = 0;
    }

    // Update hash key
    k ^= Zobrist::psq[pc][from] ^ Zobrist::psq[pc][to];

    // Reset en passant square
    if (st->epSquare != SQ_NONE)
    {
        k ^= Zobrist::enpassant[file_of(st->epSquare)];
        st->epSquare = SQ_NONE;
    }

    // Update castling rights if needed
    if (st->castlingRights && (castlingRightsMask[from] | castlingRightsMask[to]))
    {
        k ^= Zobrist::castling[st->castlingRights];
        st->castlingRights &= ~(castlingRightsMask[from] | castlingRightsMask[to]);
        k ^= Zobrist::castling[st->castlingRights];
    }

    // Move the piece. The tricky Chess960 castling is handled earlier
    if (m.type_of() != CASTLING)
    {
        dp.piece[0] = pc;
        dp.from[0]  = from;
        dp.to[0]    = to;

        move_piece(from, to);
    }

    // If the moving piece is a pawn do some special extra work
    if (type_of(pc) == PAWN)
    {
        // Set en passant square if the moved pawn can be captured
        if ((int(to) ^ int(from)) == 16
            && (pawn_attacks_bb(us, to - pawn_push(us)) & pieces(them, PAWN)))
        {
            st->epSquare = to - pawn_push(us);
            k ^= Zobrist::enpassant[file_of(st->epSquare)];
        }

        else if (m.type_of() == PROMOTION)
        {
            Piece promotion = make_piece(us, m.promotion_type());

            assert(relative_rank(us, to) == RANK_8);
            assert(type_of(promotion) >= KNIGHT && type_of(promotion) <= QUEEN);

            remove_piece(to);
            put_piece(promotion, to);

            // Promoting pawn to SQ_NONE, promoted piece from SQ_NONE
            dp.to[0]               = SQ_NONE;
            dp.piece[dp.dirty_num] = promotion;
            dp.from[dp.dirty_num]  = SQ_NONE;
            dp.to[dp.dirty_num]    = to;
            dp.dirty_num++;

            // Update hash keys
            k ^= Zobrist::psq[pc][to] ^ Zobrist::psq[promotion][to];
            st->pawnKey ^= Zobrist::psq[pc][to];
            st->materialKey ^=
              Zobrist::psq[promotion][pieceCount[promotion] - 1] ^ Zobrist::psq[pc][pieceCount[pc]];

            // Update material
            st->nonPawnMaterial[us] += PieceValue[promotion];
        }

        // Update pawn hash key
        st->pawnKey ^= Zobrist::psq[pc][from] ^ Zobrist::psq[pc][to];

        // Reset rule 50 draw counter
        st->rule50 = 0;
    }

    // Set capture piece
    st->capturedPiece = captured;

    // Update the key with the final value
    st->key = k;

    // Calculate checkers bitboard (if move gives check)
    st->checkersBB = givesCheck ? attackers_to(square<KING>(them)) & pieces(us) : 0;

    sideToMove = ~sideToMove;

    // Update king attacks used for fast check detection
    set_check_info();

    // Calculate the repetition info. It is the ply distance from the previous
    // occurrence of the same position, negative in the 3-fold case, or zero
    // if the position was not repeated.
    st->repetition = 0;
    int end        = std::min(st->rule50, st->pliesFromNull);
    if (end >= 4)
    {
        StateInfo* stp = st->previous->previous;
        for (int i = 4; i <= end; i += 2)
        {
            stp = stp->previous->previous;
            if (stp->key == st->key)
            {
                st->repetition = stp->repetition ? -i : i;
                break;
            }
        }
    }

    assert(pos_is_ok());
}

// Unmakes a move. When it returns, the position should
// be restored to exactly the same state as before the move was made.
void Position::undo_move(Move m) {

    assert(m.is_ok());

    sideToMove = ~sideToMove;

    Color  us   = sideToMove;
    Square from = m.from_sq();
    Square to   = m.to_sq();
    Piece  pc   = piece_on(to);

    assert(empty(from) || m.type_of() == CASTLING);
    assert(type_of(st->capturedPiece) != KING);

    if (m.type_of() == PROMOTION)
    {
        assert(relative_rank(us, to) == RANK_8);
        assert(type_of(pc) == m.promotion_type());
        assert(type_of(pc) >= KNIGHT && type_of(pc) <= QUEEN);

        remove_piece(to);
        pc = make_piece(us, PAWN);
        put_piece(pc, to);
    }

    if (m.type_of() == CASTLING)
    {
        Square rfrom, rto;
        do_castling<false>(us, from, to, rfrom, rto);
    }
    else
    {
        move_piece(to, from);  // Put the piece back at the source square

        if (st->capturedPiece)
        {
            Square capsq = to;

            if (m.type_of() == EN_PASSANT)
            {
                capsq -= pawn_push(us);

                assert(type_of(pc) == PAWN);
                assert(to == st->previous->epSquare);
                assert(relative_rank(us, to) == RANK_6);
                assert(piece_on(capsq) == NO_PIECE);
                assert(st->capturedPiece == make_piece(~us, PAWN));
            }

            put_piece(st->capturedPiece, capsq);  // Restore the captured piece
        }
    }

    // Finally point our state pointer back to the previous state
    st = st->previous;
    --gamePly;

    assert(pos_is_ok());
}

// Helper used to do/undo a castling move. This is a bit
// tricky in Chess960 where from/to squares can overlap.
template<bool Do>
void Position::do_castling(Color us, Square from, Square& to, Square& rfrom, Square& rto) {

    bool kingSide = to > from;
    rfrom         = to;  // Castling is encoded as "king captures friendly rook"
    rto           = relative_square(us, kingSide ? SQ_F1 : SQ_D1);
    to            = relative_square(us, kingSide ? SQ_G1 : SQ_C1);

    if (Do)
    {
        auto& dp     = st->dirtyPiece;
        dp.piece[0]  = make_piece(us, KING);
        dp.from[0]   = from;
        dp.to[0]     = to;
        dp.piece[1]  = make_piece(us, ROOK);
        dp.from[1]   = rfrom;
        dp.to[1]     = rto;
        dp.dirty_num = 2;
    }

    // Remove both pieces first since squares could overlap in Chess960
    remove_piece(Do ? from : to);
    remove_piece(Do ? rfrom : rto);
    board[Do ? from : to] = board[Do ? rfrom : rto] =
      NO_PIECE;  // remove_piece does not do this for us
    put_piece(make_piece(us, KING), Do ? to : from);
    put_piece(make_piece(us, ROOK), Do ? rto : rfrom);
}

// Used to do a "null move": it flips
// the side to move without executing any move on the board.
void Position::do_null_move(StateInfo& newSt, TranspositionTable& tt) {

    assert(!checkers());
    assert(&newSt != st);

    std::memcpy(&newSt, st, offsetof(StateInfo, accumulatorBig));

    newSt.previous = st;
    st             = &newSt;

    st->dirtyPiece.dirty_num               = 0;
    st->dirtyPiece.piece[0]                = NO_PIECE;  // Avoid checks in UpdateAccumulator()
    st->accumulatorBig.computed[WHITE]     = st->accumulatorBig.computed[BLACK] =
      st->accumulatorSmall.computed[WHITE] = st->accumulatorSmall.computed[BLACK] = false;

    if (st->epSquare != SQ_NONE)
    {
        st->key ^= Zobrist::enpassant[file_of(st->epSquare)];
        st->epSquare = SQ_NONE;
    }

    st->key ^= Zobrist::side;
    ++st->rule50;
    prefetch(tt.first_entry(key()));

    st->pliesFromNull = 0;

    sideToMove = ~sideToMove;

    set_check_info();

    st->repetition = 0;

    assert(pos_is_ok());
}

// Must be used to undo a "null move"
void Position::undo_null_move() {

    assert(!checkers());

    st         = st->previous;
    sideToMove = ~sideToMove;
}

// Computes the new hash key after the given move. Needed
// for speculative prefetch. It doesn't recognize special moves like castling,
// en passant and promotions.
Key Position::key_after(Move m) const {

    Square from     = m.from_sq();
    Square to       = m.to_sq();
    Piece  pc       = piece_on(from);
    Piece  captured = piece_on(to);
    Key    k        = st->key ^ Zobrist::side;

    if (captured)
        k ^= Zobrist::psq[captured][to];

    k ^= Zobrist::psq[pc][to] ^ Zobrist::psq[pc][from];

    return (captured || type_of(pc) == PAWN) ? k : adjust_key50<true>(k);
}

// Tests if the SEE (Static Exchange Evaluation)
// value of move is greater or equal to the given threshold. We'll use an
// algorithm similar to alpha-beta pruning with a null window.
bool Position::see_ge(Move m, int threshold) const {

    assert(m.is_ok());

    // Only deal with normal moves, assume others pass a simple SEE
    if (m.type_of() != NORMAL)
        return VALUE_ZERO >= threshold;

    Square from = m.from_sq(), to = m.to_sq();

    int swap = PieceValue[piece_on(to)] - threshold;
    if (swap < 0)
        return false;

    swap = PieceValue[piece_on(from)] - swap;
    if (swap <= 0)
        return true;

    assert(color_of(piece_on(from)) == sideToMove);
    Bitboard occupied  = pieces() ^ from ^ to;  // xoring to is important for pinned piece logic
    Color    stm       = sideToMove;
    Bitboard attackers = attackers_to(to, occupied);
    Bitboard stmAttackers, bb;
    int      res = 1;

    while (true)
    {
        stm = ~stm;
        attackers &= occupied;

        // If stm has no more attackers then give up: stm loses
        if (!(stmAttackers = attackers & pieces(stm)))
            break;

        // Don't allow pinned pieces to attack as long as there are
        // pinners on their original square.
        if (pinners(~stm) & occupied)
        {
            stmAttackers &= ~blockers_for_king(stm);

            if (!stmAttackers)
                break;
        }

        res ^= 1;

        // Locate and remove the next least valuable attacker, and add to
        // the bitboard 'attackers' any X-ray attackers behind it.
        if ((bb = stmAttackers & pieces(PAWN)))
        {
            if ((swap = PawnValue - swap) < res)
                break;
            occupied ^= least_significant_square_bb(bb);

            attackers |= attacks_bb<BISHOP>(to, occupied) & pieces(BISHOP, QUEEN);
        }

        else if ((bb = stmAttackers & pieces(KNIGHT)))
        {
            if ((swap = KnightValue - swap) < res)
                break;
            occupied ^= least_significant_square_bb(bb);
        }

        else if ((bb = stmAttackers & pieces(BISHOP)))
        {
            if ((swap = BishopValue - swap) < res)
                break;
            occupied ^= least_significant_square_bb(bb);

            attackers |= attacks_bb<BISHOP>(to, occupied) & pieces(BISHOP, QUEEN);
        }

        else if ((bb = stmAttackers & pieces(ROOK)))
        {
            if ((swap = RookValue - swap) < res)
                break;
            occupied ^= least_significant_square_bb(bb);

            attackers |= attacks_bb<ROOK>(to, occupied) & pieces(ROOK, QUEEN);
        }

        else if ((bb = stmAttackers & pieces(QUEEN)))
        {
            if ((swap = QueenValue - swap) < res)
                break;
            occupied ^= least_significant_square_bb(bb);

            attackers |= (attacks_bb<BISHOP>(to, occupied) & pieces(BISHOP, QUEEN))
                       | (attacks_bb<ROOK>(to, occupied) & pieces(ROOK, QUEEN));
        }

        else  // KING
              // If we "capture" with the king but the opponent still has attackers,
              // reverse the result.
            return (attackers & ~pieces(stm)) ? res ^ 1 : res;
    }

    return bool(res);
}

// Tests whether the position is drawn by 50-move rule
// or by repetition. It does not detect stalemates.
bool Position::is_draw(int ply) const {

    if (st->rule50 > 99 && (!checkers() || MoveList<LEGAL>(*this).size()))
        return true;

    // Return a draw score if a position repeats once earlier but strictly
    // after the root, or repeats twice before or at the root.
    return st->repetition && st->repetition < ply;
}

// Tests whether there has been at least one repetition
// of positions since the last capture or pawn move.
bool Position::has_repeated() const {

    StateInfo* stc = st;
    int        end = std::min(st->rule50, st->pliesFromNull);
    while (end-- >= 4)
    {
        if (stc->repetition)
            return true;

        stc = stc->previous;
    }
    return false;
}

// Tests if the position has a move which draws by repetition.
// This function accurately matches the outcome of is_draw() over all legal moves.
bool Position::upcoming_repetition(int ply) const {

    int j;

    int end = std::min(st->rule50, st->pliesFromNull);

    if (end < 3)
        return false;

    Key        originalKey = st->key;
    StateInfo* stp         = st->previous;
    Key        other       = originalKey ^ stp->key ^ Zobrist::side;

    for (int i = 3; i <= end; i += 2)
    {
        stp = stp->previous;
        other ^= stp->key ^ stp->previous->key ^ Zobrist::side;
        stp = stp->previous;

        if (other != 0)
            continue;

        Key moveKey = originalKey ^ stp->key;
        if ((j = H1(moveKey), cuckoo[j] == moveKey) || (j = H2(moveKey), cuckoo[j] == moveKey))
        {
            Move   move = cuckooMove[j];
            Square s1   = move.from_sq();
            Square s2   = move.to_sq();

            if (!((between_bb(s1, s2) ^ s2) & pieces()))
            {
                if (ply > i)
                    return true;

                // For nodes before or at the root, check that the move is a
                // repetition rather than a move to the current position.
                if (stp->repetition)
                    return true;
            }
        }
    }
    return false;
}

// Flips position with the white and black sides reversed. This
// is only useful for debugging e.g. for finding evaluation symmetry bugs.
void Position::flip() {

    string            f, token;
    std::stringstream ss(fen());

    for (Rank r = RANK_8; r >= RANK_1; --r)  // Piece placement
    {
        std::getline(ss, token, r > RANK_1 ? '/' : ' ');
        f.insert(0, token + (f.empty() ? " " : "/"));
    }

    ss >> token;                        // Active color
    f += (token == "w" ? "B " : "W ");  // Will be lowercased later

    ss >> token;  // Castling availability
    f += token + " ";

    std::transform(f.begin(), f.end(), f.begin(),
                   [](char c) { return char(islower(c) ? toupper(c) : tolower(c)); });

    ss >> token;  // En passant square
    f += (token == "-" ? token : token.replace(1, 1, token[1] == '3' ? "6" : "3"));

    std::getline(ss, token);  // Half and full moves
    f += token;

    set(f, is_chess960(), st);

    assert(pos_is_ok());
}

// Performs some consistency checks for the position object
// and raise an assert if something wrong is detected.
// This is meant to be helpful when debugging.
bool Position::pos_is_ok() const {

    constexpr bool Fast = true;  // Quick (default) or full check?

    if ((sideToMove != WHITE && sideToMove != BLACK) || piece_on(square<KING>(WHITE)) != W_KING
        || piece_on(square<KING>(BLACK)) != B_KING
        || (ep_square() != SQ_NONE && relative_rank(sideToMove, ep_square()) != RANK_6))
        assert(0 && "pos_is_ok: Default");

    if (Fast)
        return true;

    if (pieceCount[W_KING] != 1 || pieceCount[B_KING] != 1
        || attackers_to(square<KING>(~sideToMove)) & pieces(sideToMove))
        assert(0 && "pos_is_ok: Kings");

    if ((pieces(PAWN) & (Rank1BB | Rank8BB)) || pieceCount[W_PAWN] > 8 || pieceCount[B_PAWN] > 8)
        assert(0 && "pos_is_ok: Pawns");

    if ((pieces(WHITE) & pieces(BLACK)) || (pieces(WHITE) | pieces(BLACK)) != pieces()
        || popcount(pieces(WHITE)) > 16 || popcount(pieces(BLACK)) > 16)
        assert(0 && "pos_is_ok: Bitboards");

    for (PieceType p1 = PAWN; p1 <= KING; ++p1)
        for (PieceType p2 = PAWN; p2 <= KING; ++p2)
            if (p1 != p2 && (pieces(p1) & pieces(p2)))
                assert(0 && "pos_is_ok: Bitboards");

    for (Piece pc : Pieces)
        if (pieceCount[pc] != popcount(pieces(color_of(pc), type_of(pc)))
            || pieceCount[pc] != std::count(board, board + SQUARE_NB, pc))
            assert(0 && "pos_is_ok: Pieces");

    for (Color c : {WHITE, BLACK})
        for (CastlingRights cr : {c & KING_SIDE, c & QUEEN_SIDE})
        {
            if (!can_castle(cr))
                continue;

            if (piece_on(castlingRookSquare[cr]) != make_piece(c, ROOK)
                || castlingRightsMask[castlingRookSquare[cr]] != cr
                || (castlingRightsMask[square<KING>(c)] & cr) != cr)
                assert(0 && "pos_is_ok: Castling");
        }

    return true;
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace {

// clang-format off
const std::vector<std::string> Defaults = {
  "setoption name UCI_Chess960 value false",
  "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1",
  "r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 10",
  "8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - 0 11",
  "4rrk1/pp1n3p/3q2pQ/2p1pb2/2PP4/2P3N1/P2B2PP/4RRK1 b - - 7 19",
  "rq3rk1/ppp2ppp/1bnpb3/3N2B1/3NP3/7P/PPPQ1PP1/2KR3R w - - 7 14 moves d4e6",
  "r1bq1r1k/1pp1n1pp/1p1p4/4p2Q/4Pp2/1BNP4/PPP2PPP/3R1RK1 w - - 2 14 moves g2g4",
  "r3r1k1/2p2ppp/p1p1bn2/8/1q2P3/2NPQN2/PPP3PP/R4RK1 b - - 2 15",
  "r1bbk1nr/pp3p1p/2n5/1N4p1/2Np1B2/8/PPP2PPP/2KR1B1R w kq - 0 13",
  "r1bq1rk1/ppp1nppp/4n3/3p3Q/3P4/1BP1B3/PP1N2PP/R4RK1 w - - 1 16",
  "4r1k1/r1q2ppp/ppp2n2/4P3/5Rb1/1N1BQ3/PPP3PP/R5K1 w - - 1 17",
  "2rqkb1r/ppp2p2/2npb1p1/1N1Nn2p/2P1PP2/8/PP2B1PP/R1BQK2R b KQ - 0 11",
  "r1bq1r1k/b1p1npp1/p2p3p/1p6/3PP3/1B2NN2/PP3PPP/R2Q1RK1 w - - 1 16",
  "3r1rk1/p5pp/bpp1pp2/8/q1PP1P2/b3P3/P2NQRPP/1R2B1K1 b - - 6 22",
  "r1q2rk1/2p1bppp/2Pp4/p6b/Q1PNp3/4B3/PP1R1PPP/2K4R w - - 2 18",
  "4k2r/1pb2ppp/1p2p3/1R1p4/3P4/2r1PN2/P4PPP/1R4K1 b - - 3 22",
  "3q2k1/pb3p1p/4pbp1/2r5/PpN2N2/1P2P2P/5PP1/Q2R2K1 b - - 4 26",
  "6k1/6p1/6Pp/ppp5/3pn2P/1P3K2/1PP2P2/3N4 b - - 0 1",
  "3b4/5kp1/1p1p1p1p/pP1PpP1P/P1P1P3/3KN3/8/8 w - - 0 1",
  "2K5/p7/7P/5pR1/8/5k2/r7/8 w - - 0 1 moves g5g6 f3e3 g6g5 e3f3",
  "8/6pk/1p6/8/PP3p1p/5P2/4KP1q/3Q4 w - - 0 1",
  "7k/3p2pp/4q3/8/4Q3/5Kp1/P6b/8 w - - 0 1",
  "8/2p5/8/2kPKp1p/2p4P/2P5/3P4/8 w - - 0 1",
  "8/1p3pp1/7p/5P1P/2k3P1/8/2K2P2/8 w - - 0 1",
  "8/pp2r1k1/2p1p3/3pP2p/1P1P1P1P/P5KR/8/8 w - - 0 1",
  "8/3p4/p1bk3p/Pp6/1Kp1PpPp/2P2P1P/2P5/5B2 b - - 0 1",
  "5k2/7R/4P2p/5K2/p1r2P1p/8/8/8 b - - 0 1",
  "6k1/6p1/P6p/r1N5/5p2/7P/1b3PP1/4R1K1 w - - 0 1",
  "1r3k2/4q3/2Pp3b/3Bp3/2Q2p2/1p1P2P1/1P2KP2/3N4 w - - 0 1",
  "6k1/4pp1p/3p2p1/P1pPb3/R7/1r2P1PP/3B1P2/6K1 w - - 0 1",
  "8/3p3B/5p2/5P2/p7/PP5b/k7/6K1 w - - 0 1",
  "5rk1/q6p/2p3bR/1pPp1rP1/1P1Pp3/P3B1Q1/1K3P2/R7 w - - 93 90",
  "4rrk1/1p1nq3/p7/2p1P1pp/3P2bp/3Q1Bn1/PPPB4/1K2R1NR w - - 40 21",
  "r3k2r/3nnpbp/q2pp1p1/p7/Pp1PPPP1/4BNN1/1P5P/R2Q1RK1 w kq - 0 16",
  "3Qb1k1/1r2ppb1/pN1n2q1/Pp1Pp1Pr/4P2p/4BP2/4B1R1/1R5K b - - 11 40",
  "4k3/3q1r2/1N2r1b1/3ppN2/2nPP3/1B1R2n1/2R1Q3/3K4 w - - 5 1",

  // 5-man positions
  "8/8/8/8/5kp1/P7/8/1K1N4 w - - 0 1",     // Kc2 - mate
  "8/8/8/5N2/8/p7/8/2NK3k w - - 0 1",      // Na2 - mate
  "8/3k4/8/8/8/4B3/4KB2/2B5 w - - 0 1",    // draw

  // 6-man positions
  "8/8/1P6/5pr1/8/4R3/7k/2K5 w - - 0 1",   // Re5 - mate
  "8/2p4P/8/kr6/6R1/8/8/1K6 w - - 0 1",    // Ka2 - mate
  "8/8/3P3k/8/1p6/8/1P6/1K3n2 b - - 0 1",  // Nd2 - draw

  // 7-man positions
  "8/R7/2q5/8/6k1/8/1P5p/K6R w - - 0 124", // Draw

  // Mate and stalemate positions
  "6k1/3b3r/1p1p4/p1n2p2/1PPNpP1q/P3Q1p1/1R1RB1P1/5K2 b - - 0 1",
  "r2r1n2/pp2bk2/2p1p2p/3q4/3PN1QP/2P3R1/P4PP1/5RK1 w - - 0 1",
  "8/8/8/8/8/6k1/6p1/6K1 w - -",
  "7k/7P/6K1/8/3B4/8/8/8 b - -",

  // Chess 960
  "setoption name UCI_Chess960 value true",
  "bbqnnrkr/pppppppp/8/8/8/8/PPPPPPPP/BBQNNRKR w HFhf - 0 1 moves g2g3 d7d5 d2d4 c8h3 c1g5 e8d6 g5e7 f7f6",
  "nqbnrkrb/pppppppp/8/8/8/8/PPPPPPPP/NQBNRKRB w KQkq - 0 1",
  "setoption name UCI_Chess960 value false"
};
// clang-format on

}  // namespace

namespace Stockfish::Benchmark {

// Builds a list of UCI commands to be run by bench. There
// are five parameters: TT size in MB, number of search threads that
// should be used, the limit value spent for each position, a file name
// where to look for positions in FEN format, and the type of the limit:
// depth, perft, nodes and movetime (in milliseconds). Examples:
//
// bench                            : search default positions up to depth 13
// bench 64 1 15                    : search default positions up to depth 15 (TT = 64MB)
// bench 64 1 100000 default nodes  : search default positions for 100K nodes each
// bench 64 4 5000 current movetime : search current position with 4 threads for 5 sec
// bench 16 1 5 blah perft          : run a perft 5 on positions in file "blah"
std::vector<std::string> setup_bench(const std::string& currentFen, std::istream& is) {

    std::vector<std::string> fens, list;
    std::string              go, token;

    // Assign default values to missing arguments
    std::string ttSize    = (is >> token) ? token : "16";
    std::string threads   = (is >> token) ? token : "1";
    std::string limit     = (is >> token) ? token : "13";
    std::string fenFile   = (is >> token) ? token : "default";
    std::string limitType = (is >> token) ? token : "depth";

    go = limitType == "eval" ? "eval" : "go " + limitType + " " + limit;

    if (fenFile == "default")
        fens = Defaults;

    else if (fenFile == "current")
        fens.push_back(currentFen);

    else
    {
        std::string   fen;
        std::ifstream file(fenFile);

        if (!file.is_open())
        {
            std::cerr << "Unable to open file " << fenFile << std::endl;
            exit(EXIT_FAILURE);
        }

        while (getline(file, fen))
            if (!fen.empty())
                fens.push_back(fen);

        file.close();
    }

    list.emplace_back("setoption name Threads value " + threads);
    list.emplace_back("setoption name Hash value " + ttSize);
    list.emplace_back("ucinewgame");

    for (const std::string& fen : fens)
        if (fen.find("setoption") != std::string::npos)
            list.emplace_back(fen);
        else
        {
            list.emplace_back("position fen " + fen);
            list.emplace_back(go);
        }

    return list;
}

}  // namespace Stockfish
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

uint8_t PopCnt16[1 << 16];
uint8_t SquareDistance[SQUARE_NB][SQUARE_NB];

Bitboard LineBB[SQUARE_NB][SQUARE_NB];
Bitboard BetweenBB[SQUARE_NB][SQUARE_NB];
Bitboard PseudoAttacks[PIECE_TYPE_NB][SQUARE_NB];
Bitboard PawnAttacks[COLOR_NB][SQUARE_NB];

Magic RookMagics[SQUARE_NB];
Magic BishopMagics[SQUARE_NB];

namespace {

Bitboard RookTable[0x19000];   // To store rook attacks
Bitboard BishopTable[0x1480];  // To store bishop attacks

void init_magics(PieceType pt, Bitboard table[], Magic magics[]);

// Returns the bitboard of target square for the given step
// from the given square. If the step is off the board, returns empty bitboard.
Bitboard safe_destination(Square s, int step) {
    Square to = Square(s + step);
    return is_ok(to) && distance(s, to) <= 2 ? square_bb(to) : Bitboard(0);
}
}

// Returns an ASCII representation of a bitboard suitable
// to be printed to standard output. Useful for debugging.
std::string Bitboards::pretty(Bitboard b) {

    std::string s = "+---+---+---+---+---+---+---+---+\n";

    for (Rank r = RANK_8; r >= RANK_1; --r)
    {
        for (File f = FILE_A; f <= FILE_H; ++f)
            s += b & make_square(f, r) ? "| X " : "|   ";
        s += "| " + std::to_string(1 + r) + "\n+---+---+---+---+---+---+---+---+\n";
    }
    s += "  a   b   c   d   e   f   g   h\n";

    return s;
}

// Initializes various bitboard tables. It is called at
// startup and relies on global objects to be already zero-initialized.
void Bitboards::init() {

    for (unsigned i = 0; i < (1 << 16); ++i)
        PopCnt16[i] = uint8_t(std::bitset<16>(i).count());

    for (Square s1 = SQ_A1; s1 <= SQ_H8; ++s1)
        for (Square s2 = SQ_A1; s2 <= SQ_H8; ++s2)
            SquareDistance[s1][s2] = std::max(distance<File>(s1, s2), distance<Rank>(s1, s2));

    init_magics(ROOK, RookTable, RookMagics);
    init_magics(BISHOP, BishopTable, BishopMagics);

    for (Square s1 = SQ_A1; s1 <= SQ_H8; ++s1)
    {
        PawnAttacks[WHITE][s1] = pawn_attacks_bb<WHITE>(square_bb(s1));
        PawnAttacks[BLACK][s1] = pawn_attacks_bb<BLACK>(square_bb(s1));

        for (int step : {-9, -8, -7, -1, 1, 7, 8, 9})
            PseudoAttacks[KING][s1] |= safe_destination(s1, step);

        for (int step : {-17, -15, -10, -6, 6, 10, 15, 17})
            PseudoAttacks[KNIGHT][s1] |= safe_destination(s1, step);

        PseudoAttacks[QUEEN][s1] = PseudoAttacks[BISHOP][s1] = attacks_bb<BISHOP>(s1, 0);
        PseudoAttacks[QUEEN][s1] |= PseudoAttacks[ROOK][s1]  = attacks_bb<ROOK>(s1, 0);

        for (PieceType pt : {BISHOP, ROOK})
            for (Square s2 = SQ_A1; s2 <= SQ_H8; ++s2)
            {
                if (PseudoAttacks[pt][s1] & s2)
                {
                    LineBB[s1][s2] = (attacks_bb(pt, s1, 0) & attacks_bb(pt, s2, 0)) | s1 | s2;
                    BetweenBB[s1][s2] =
                      (attacks_bb(pt, s1, square_bb(s2)) & attacks_bb(pt, s2, square_bb(s1)));
                }
                BetweenBB[s1][s2] |= s2;
            }
    }
}

namespace {

Bitboard sliding_attack(PieceType pt, Square sq, Bitboard occupied) {

    Bitboard  attacks             = 0;
    Direction RookDirections[4]   = {NORTH, SOUTH, EAST, WEST};
    Direction BishopDirections[4] = {NORTH_EAST, SOUTH_EAST, SOUTH_WEST, NORTH_WEST};

    for (Direction d : (pt == ROOK ? RookDirections : BishopDirections))
    {
        Square s = sq;
        while (safe_destination(s, d))
        {
            attacks |= (s += d);
            if (occupied & s)
            {
                break;
            }
        }
    }

    return attacks;
}

// Computes all rook and bishop attacks at startup. Magic
// bitboards are used to look up attacks of sliding pieces. As a reference see
// www.chessprogramming.org/Magic_Bitboards. In particular, here we use the so
// called "fancy" approach.
void init_magics(PieceType pt, Bitboard table[], Magic magics[]) {

    // Optimal PRNG seeds to pick the correct magics in the shortest time
    int seeds[][RANK_NB] = {{8977, 44560, 54343, 38998, 5731, 95205, 104912, 17020},
                            {728, 10316, 55013, 32803, 12281, 15100, 16645, 255}};

    Bitboard occupancy[4096], reference[4096], edges, b;
    int      epoch[4096] = {}, cnt = 0, size = 0;

    for (Square s = SQ_A1; s <= SQ_H8; ++s)
    {
        // Board edges are not considered in the relevant occupancies
        edges = ((Rank1BB | Rank8BB) & ~rank_bb(s)) | ((FileABB | FileHBB) & ~file_bb(s));

        // Given a square 's', the mask is the bitboard of sliding attacks from
        // 's' computed on an empty board. The index must be big enough to contain
        // all the attacks for each possible subset of the mask and so is 2 power
        // the number of 1s of the mask. Hence we deduce the size of the shift to
        // apply to the 64 or 32 bits word to get the index.
        Magic& m = magics[s];
        m.mask   = sliding_attack(pt, s, 0) & ~edges;
        m.shift  = (Is64Bit ? 64 : 32) - popcount(m.mask);

        // Set the offset for the attacks table of the square. We have individual
        // table sizes for each square with "Fancy Magic Bitboards".
        m.attacks = s == SQ_A1 ? table : magics[s - 1].attacks + size;

        // Use Carry-Rippler trick to enumerate all subsets of masks[s] and
        // store the corresponding sliding attack bitboard in reference[].
        b = size = 0;
        do
        {
            occupancy[size] = b;
            reference[size] = sliding_attack(pt, s, b);

            if (HasPext)
                m.attacks[pext(b, m.mask)] = reference[size];

            size++;
            b = (b - m.mask) & m.mask;
        } while (b);

        if (HasPext)
            continue;

        PRNG rng(seeds[Is64Bit][rank_of(s)]);

        // Find a magic for square 's' picking up an (almost) random number
        // until we find the one that passes the verification test.
        for (int i = 0; i < size;)
        {
            for (m.magic = 0; popcount((m.magic * m.mask) >> 56) < 6;)
                m.magic = rng.sparse_rand<Bitboard>();

            // A good magic must map every possible occupancy to an index that
            // looks up the correct sliding attack in the attacks[s] database.
            // Note that we build up the database for square 's' as a side
            // effect of verifying the magic. Keep track of the attempt count
            // and save it in epoch[], little speed-up trick to avoid resetting
            // m.attacks[] after every failed attempt.
            for (++cnt, i = 0; i < size; ++i)
            {
                unsigned idx = m.index(occupancy[i]);

                if (epoch[idx] < cnt)
                {
                    epoch[idx]     = cnt;
                    m.attacks[idx] = reference[i];
                }
                else if (m.attacks[idx] != reference[i])
                    break;
            }
        }
    }
}
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

// Returns a static, purely materialistic evaluation of the position from
// the point of view of the given color. It can be divided by PawnValue to get
// an approximation of the material advantage on the board in terms of pawns.
int Eval::simple_eval(const Position& pos, Color c) {
    return PawnValue * (pos.count<PAWN>(c) - pos.count<PAWN>(~c))
         + (pos.non_pawn_material(c) - pos.non_pawn_material(~c));
}

bool Eval::use_smallnet(const Position& pos) {
    int simpleEval = simple_eval(pos, pos.side_to_move());
    return std::abs(simpleEval) > 962;
}

// Evaluate is the evaluator for the outer world. It returns a static evaluation
// of the position from the point of view of the side to move.
Value Eval::evaluate(const Eval::NNUE::Networks&    networks,
                     const Position&                pos,
                     Eval::NNUE::AccumulatorCaches& caches,
                     int                            optimism) {

    assert(!pos.checkers());

    bool smallNet = use_smallnet(pos);
    int  v;

    auto [psqt, positional] = smallNet ? networks.small.evaluate(pos, &caches.small)
                                       : networks.big.evaluate(pos, &caches.big);

    Value nnue = (125 * psqt + 131 * positional) / 128;

    // Re-evaluate the position when higher eval accuracy is worth the time spent
    if (smallNet && (nnue * psqt < 0 || std::abs(nnue) < 227))
    {
        std::tie(psqt, positional) = networks.big.evaluate(pos, &caches.big);
        nnue                       = (125 * psqt + 131 * positional) / 128;
        smallNet                   = false;
    }

    // Blend optimism and eval with nnue complexity
    int nnueComplexity = std::abs(psqt - positional);
    optimism += optimism * nnueComplexity / (smallNet ? 433 : 453);
    nnue -= nnue * nnueComplexity / (smallNet ? 18815 : 17864);

    int material = (smallNet ? 553 : 532) * pos.count<PAWN>() + pos.non_pawn_material();
    v = (nnue * (73921 + material) + optimism * (8112 + material)) / (smallNet ? 68104 : 74715);

    // Evaluation grain (to get more alpha-beta cuts) with randomization (for robustness)
    v = (v / 16) * 16 - 1 + (pos.key() & 0x2);

    // Damp down the evaluation linearly when shuffling
    v -= v * pos.rule50_count() / 212;

    // Guarantee evaluation does not hit the tablebase range
    v = std::clamp(v, VALUE_TB_LOSS_IN_MAX_PLY + 1, VALUE_TB_WIN_IN_MAX_PLY - 1);

    return v;
}

// Like evaluate(), but instead of returning a value, it returns
// a string (suitable for outputting to stdout) that contains the detailed
// descriptions and values of each evaluation term. Useful for debugging.
// Trace scores are from white's point of view
std::string Eval::trace(Position& pos, const Eval::NNUE::Networks& networks) {

    if (pos.checkers())
        return "Final evaluation: none (in check)";

    auto caches = std::make_unique<Eval::NNUE::AccumulatorCaches>(networks);

    std::stringstream ss;
    ss << std::showpoint << std::noshowpos << std::fixed << std::setprecision(2);
    ss << '\n' << NNUE::trace(pos, networks, *caches) << '\n';

    ss << std::showpoint << std::showpos << std::fixed << std::setprecision(2) << std::setw(15);

    auto [psqt, positional] = networks.big.evaluate(pos, &caches->big);
    Value v                 = psqt + positional;
    v                       = pos.side_to_move() == WHITE ? v : -v;
    ss << "NNUE evaluation        " << 0.01 * UCIEngine::to_cp(v, pos) << " (white side)\n";

    v = evaluate(networks, pos, *caches, VALUE_ZERO);
    v = pos.side_to_move() == WHITE ? v : -v;
    ss << "Final evaluation       " << 0.01 * UCIEngine::to_cp(v, pos) << " (white side)";
    ss << " [with scaled NNUE, ...]";
    ss << "\n";

    return ss.str();
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

namespace {

// Version number or dev.
constexpr std::string_view version = "17";

// Our fancy logging facility. The trick here is to replace cin.rdbuf() and
// cout.rdbuf() with two Tie objects that tie cin and cout to a file stream. We
// can toggle the logging of std::cout and std:cin at runtime whilst preserving
// usual I/O functionality, all without changing a single line of code!
// Idea from http://groups.google.com/group/comp.lang.c++/msg/1d941c0f26ea0d81

struct Tie: public std::streambuf {  // MSVC requires split streambuf for cin and cout

    Tie(std::streambuf* b, std::streambuf* l) :
        buf(b),
        logBuf(l) {}

    int sync() override { return logBuf->pubsync(), buf->pubsync(); }
    int overflow(int c) override { return log(buf->sputc(char(c)), "<< "); }
    int underflow() override { return buf->sgetc(); }
    int uflow() override { return log(buf->sbumpc(), ">> "); }

    std::streambuf *buf, *logBuf;

    int log(int c, const char* prefix) {

        static int last = '\n';  // Single log file

        if (last == '\n')
            logBuf->sputn(prefix, 3);

        return last = logBuf->sputc(char(c));
    }
};

class Logger {

    Logger() :
        in(std::cin.rdbuf(), file.rdbuf()),
        out(std::cout.rdbuf(), file.rdbuf()) {}
    ~Logger() { start(""); }

    std::ofstream file;
    Tie           in, out;

   public:
    static void start(const std::string& fname) {

        static Logger l;

        if (l.file.is_open())
        {
            std::cout.rdbuf(l.out.buf);
            std::cin.rdbuf(l.in.buf);
            l.file.close();
        }

        if (!fname.empty())
        {
            l.file.open(fname, std::ifstream::out);

            if (!l.file.is_open())
            {
                std::cerr << "Unable to open debug log file " << fname << std::endl;
                exit(EXIT_FAILURE);
            }

            std::cin.rdbuf(&l.in);
            std::cout.rdbuf(&l.out);
        }
    }
};

}  // namespace

// Returns the full name of the current Stockfish version.
//
// For local dev compiles we try to append the commit SHA and
// commit date from git. If that fails only the local compilation
// date is set and "nogit" is specified:
//      Stockfish dev-YYYYMMDD-SHA
//      or
//      Stockfish dev-YYYYMMDD-nogit
//
// For releases (non-dev builds) we only include the version number:
//      Stockfish version
std::string engine_info(bool to_uci) {
    std::stringstream ss;
    ss << "Stockfish " << version << std::setfill('0');

    if constexpr (version == "dev")
    {
        ss << "-";
#ifdef GIT_DATE
        ss << stringify(GIT_DATE);
#else
        constexpr std::string_view months("Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec");

        std::string       month, day, year;
        std::stringstream date(__DATE__);  // From compiler, format is "Sep 21 2008"

        date >> month >> day >> year;
        ss << year << std::setw(2) << std::setfill('0') << (1 + months.find(month) / 4)
           << std::setw(2) << std::setfill('0') << day;
#endif

        ss << "-";

#ifdef GIT_SHA
        ss << stringify(GIT_SHA);
#else
        ss << "nogit";
#endif
    }

    ss << (to_uci ? "\nid author " : " by ") << "the Stockfish developers (see AUTHORS file)";

    return ss.str();
}

// Returns a string trying to describe the compiler we use
std::string compiler_info() {

#define make_version_string(major, minor, patch) \
    stringify(major) "." stringify(minor) "." stringify(patch)

    // Predefined macros hell:
    //
    // __GNUC__                Compiler is GCC, Clang or ICX
    // __clang__               Compiler is Clang or ICX
    // __INTEL_LLVM_COMPILER   Compiler is ICX
    // _MSC_VER                Compiler is MSVC
    // _WIN32                  Building on Windows (any)
    // _WIN64                  Building on Windows 64 bit

    std::string compiler = "\nCompiled by                : ";

#if defined(__INTEL_LLVM_COMPILER)
    compiler += "ICX ";
    compiler += stringify(__INTEL_LLVM_COMPILER);
#elif defined(__clang__)
    compiler += "clang++ ";
    compiler += make_version_string(__clang_major__, __clang_minor__, __clang_patchlevel__);
#elif _MSC_VER
    compiler += "MSVC ";
    compiler += "(version ";
    compiler += stringify(_MSC_FULL_VER) "." stringify(_MSC_BUILD);
    compiler += ")";
#elif defined(__e2k__) && defined(__LCC__)
    #define dot_ver2(n) \
        compiler += char('.'); \
        compiler += char('0' + (n) / 10); \
        compiler += char('0' + (n) % 10);

    compiler += "MCST LCC ";
    compiler += "(version ";
    compiler += std::to_string(__LCC__ / 100);
    dot_ver2(__LCC__ % 100) dot_ver2(__LCC_MINOR__) compiler += ")";
#elif __GNUC__
    compiler += "g++ (GNUC) ";
    compiler += make_version_string(__GNUC__, __GNUC_MINOR__, __GNUC_PATCHLEVEL__);
#else
    compiler += "Unknown compiler ";
    compiler += "(unknown version)";
#endif

#if defined(__APPLE__)
    compiler += " on Apple";
#elif defined(__CYGWIN__)
    compiler += " on Cygwin";
#elif defined(__MINGW64__)
    compiler += " on MinGW64";
#elif defined(__MINGW32__)
    compiler += " on MinGW32";
#elif defined(__ANDROID__)
    compiler += " on Android";
#elif defined(__linux__)
    compiler += " on Linux";
#elif defined(_WIN64)
    compiler += " on Microsoft Windows 64-bit";
#elif defined(_WIN32)
    compiler += " on Microsoft Windows 32-bit";
#else
    compiler += " on unknown system";
#endif

    compiler += "\nCompilation architecture   : ";
#if defined(ARCH)
    compiler += stringify(ARCH);
#else
    compiler += "(undefined architecture)";
#endif

    compiler += "\nCompilation settings       : ";
    compiler += (Is64Bit ? "64bit" : "32bit");
#if defined(USE_VNNI)
    compiler += " VNNI";
#endif
#if defined(USE_AVX512)
    compiler += " AVX512";
#endif
    compiler += (HasPext ? " BMI2" : "");
#if defined(USE_AVX2)
    compiler += " AVX2";
#endif
#if defined(USE_SSE41)
    compiler += " SSE41";
#endif
#if defined(USE_SSSE3)
    compiler += " SSSE3";
#endif
#if defined(USE_SSE2)
    compiler += " SSE2";
#endif
    compiler += (HasPopCnt ? " POPCNT" : "");
#if defined(USE_NEON_DOTPROD)
    compiler += " NEON_DOTPROD";
#elif defined(USE_NEON)
    compiler += " NEON";
#endif

#if !defined(NDEBUG)
    compiler += " DEBUG";
#endif

    compiler += "\nCompiler __VERSION__ macro : ";
#ifdef __VERSION__
    compiler += __VERSION__;
#else
    compiler += "(undefined macro)";
#endif

    compiler += "\n";

    return compiler;
}

// Debug functions used mainly to collect run-time statistics
constexpr int MaxDebugSlots = 32;

namespace {

template<size_t N>
struct DebugInfo {
    std::atomic<int64_t> data[N] = {0};

    constexpr std::atomic<int64_t>& operator[](int index) { return data[index]; }
};

struct DebugExtremes: public DebugInfo<3> {
    DebugExtremes() {
        data[1] = std::numeric_limits<int64_t>::min();
        data[2] = std::numeric_limits<int64_t>::max();
    }
};

DebugInfo<2>  hit[MaxDebugSlots];
DebugInfo<2>  mean[MaxDebugSlots];
DebugInfo<3>  stdev[MaxDebugSlots];
DebugInfo<6>  correl[MaxDebugSlots];
DebugExtremes extremes[MaxDebugSlots];

}  // namespace

void dbg_hit_on(bool cond, int slot) {

    ++hit[slot][0];
    if (cond)
        ++hit[slot][1];
}

void dbg_mean_of(int64_t value, int slot) {

    ++mean[slot][0];
    mean[slot][1] += value;
}

void dbg_stdev_of(int64_t value, int slot) {

    ++stdev[slot][0];
    stdev[slot][1] += value;
    stdev[slot][2] += value * value;
}

void dbg_extremes_of(int64_t value, int slot) {
    ++extremes[slot][0];

    int64_t current_max = extremes[slot][1].load();
    while (current_max < value && !extremes[slot][1].compare_exchange_weak(current_max, value))
    {}

    int64_t current_min = extremes[slot][2].load();
    while (current_min > value && !extremes[slot][2].compare_exchange_weak(current_min, value))
    {}
}

void dbg_correl_of(int64_t value1, int64_t value2, int slot) {

    ++correl[slot][0];
    correl[slot][1] += value1;
    correl[slot][2] += value1 * value1;
    correl[slot][3] += value2;
    correl[slot][4] += value2 * value2;
    correl[slot][5] += value1 * value2;
}

void dbg_print() {

    int64_t n;
    auto    E   = [&n](int64_t x) { return double(x) / n; };
    auto    sqr = [](double x) { return x * x; };

    for (int i = 0; i < MaxDebugSlots; ++i)
        if ((n = hit[i][0]))
            std::cerr << "Hit #" << i << ": Total " << n << " Hits " << hit[i][1]
                      << " Hit Rate (%) " << 100.0 * E(hit[i][1]) << std::endl;

    for (int i = 0; i < MaxDebugSlots; ++i)
        if ((n = mean[i][0]))
        {
            std::cerr << "Mean #" << i << ": Total " << n << " Mean " << E(mean[i][1]) << std::endl;
        }

    for (int i = 0; i < MaxDebugSlots; ++i)
        if ((n = stdev[i][0]))
        {
            double r = sqrt(E(stdev[i][2]) - sqr(E(stdev[i][1])));
            std::cerr << "Stdev #" << i << ": Total " << n << " Stdev " << r << std::endl;
        }

    for (int i = 0; i < MaxDebugSlots; ++i)
        if ((n = extremes[i][0]))
        {
            std::cerr << "Extremity #" << i << ": Total " << n << " Min " << extremes[i][2]
                      << " Max " << extremes[i][1] << std::endl;
        }

    for (int i = 0; i < MaxDebugSlots; ++i)
        if ((n = correl[i][0]))
        {
            double r = (E(correl[i][5]) - E(correl[i][1]) * E(correl[i][3]))
                     / (sqrt(E(correl[i][2]) - sqr(E(correl[i][1])))
                        * sqrt(E(correl[i][4]) - sqr(E(correl[i][3]))));
            std::cerr << "Correl. #" << i << ": Total " << n << " Coefficient " << r << std::endl;
        }
}

// Used to serialize access to std::cout
// to avoid multiple threads writing at the same time.
std::ostream& operator<<(std::ostream& os, SyncCout sc) {

    static std::mutex m;

    if (sc == IO_LOCK)
        m.lock();

    if (sc == IO_UNLOCK)
        m.unlock();

    return os;
}

void sync_cout_start() { std::cout << IO_LOCK; }
void sync_cout_end() { std::cout << IO_UNLOCK; }

// Trampoline helper to avoid moving Logger to misc.h
void start_logger(const std::string& fname) { Logger::start(fname); }

#ifdef NO_PREFETCH

void prefetch(const void*) {}

#else

void prefetch(const void* addr) {

    #if defined(_MSC_VER)
    _mm_prefetch((char const*) addr, _MM_HINT_T0);
    #else
    __builtin_prefetch(addr);
    #endif
}

#endif

#ifdef _WIN32
    #define GETCWD _getcwd
#else
    #define GETCWD getcwd
#endif

size_t str_to_size_t(const std::string& s) {
    unsigned long long value = std::stoull(s);
    if (value > std::numeric_limits<size_t>::max())
        std::exit(EXIT_FAILURE);
    return static_cast<size_t>(value);
}

std::optional<std::string> read_file_to_string(const std::string& path) {
    std::ifstream f(path, std::ios_base::binary);
    if (!f)
        return std::nullopt;
    return std::string(std::istreambuf_iterator<char>(f), std::istreambuf_iterator<char>());
}

void remove_whitespace(std::string& s) {
    s.erase(std::remove_if(s.begin(), s.end(), [](char c) { return std::isspace(c); }), s.end());
}

bool is_whitespace(const std::string& s) {
    return std::all_of(s.begin(), s.end(), [](char c) { return std::isspace(c); });
}

std::string CommandLine::get_binary_directory(std::string argv0) {
    std::string pathSeparator;

#ifdef _WIN32
    pathSeparator = "\\";
    #ifdef _MSC_VER
    // Under windows "GreenChess" may not have the extension. Also _get_pgmptr() had
    // issues in some Windows 10 versions, so check returned values carefully.
    char* pgmptr = nullptr;
    if (!_get_pgmptr(&pgmptr) && pgmptr != nullptr && *pgmptr)
        argv0 = pgmptr;
    #endif
#else
    pathSeparator = "/";
#endif

    // Extract the working directory
    auto workingDirectory = CommandLine::get_working_directory();

    // Extract the binary directory path from argv0
    auto   binaryDirectory = argv0;
    size_t pos             = binaryDirectory.find_last_of("\\/");
    if (pos == std::string::npos)
        binaryDirectory = "." + pathSeparator;
    else
        binaryDirectory.resize(pos + 1);

    // Pattern replacement: "./" at the start of path is replaced by the working directory
    if (binaryDirectory.find("." + pathSeparator) == 0)
        binaryDirectory.replace(0, 1, workingDirectory);

    return binaryDirectory;
}

std::string CommandLine::get_working_directory() {
    std::string workingDirectory = "";
    char        buff[40000];
    char*       cwd = GETCWD(buff, 40000);
    if (cwd)
        workingDirectory = cwd;

    return workingDirectory;
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

namespace {

template<GenType Type, Direction D, bool Enemy>
ExtMove* make_promotions(ExtMove* moveList, [[maybe_unused]] Square to) {

    constexpr bool all = Type == EVASIONS || Type == NON_EVASIONS;

    if constexpr (Type == CAPTURES || all)
        *moveList++ = Move::make<PROMOTION>(to - D, to, QUEEN);

    if constexpr ((Type == CAPTURES && Enemy) || (Type == QUIETS && !Enemy) || all)
    {
        *moveList++ = Move::make<PROMOTION>(to - D, to, ROOK);
        *moveList++ = Move::make<PROMOTION>(to - D, to, BISHOP);
        *moveList++ = Move::make<PROMOTION>(to - D, to, KNIGHT);
    }

    return moveList;
}

template<Color Us, GenType Type>
ExtMove* generate_pawn_moves(const Position& pos, ExtMove* moveList, Bitboard target) {

    constexpr Color     Them     = ~Us;
    constexpr Bitboard  TRank7BB = (Us == WHITE ? Rank7BB : Rank2BB);
    constexpr Bitboard  TRank3BB = (Us == WHITE ? Rank3BB : Rank6BB);
    constexpr Direction Up       = pawn_push(Us);
    constexpr Direction UpRight  = (Us == WHITE ? NORTH_EAST : SOUTH_WEST);
    constexpr Direction UpLeft   = (Us == WHITE ? NORTH_WEST : SOUTH_EAST);

    const Bitboard emptySquares = ~pos.pieces();
    const Bitboard enemies      = Type == EVASIONS ? pos.checkers() : pos.pieces(Them);

    Bitboard pawnsOn7    = pos.pieces(Us, PAWN) & TRank7BB;
    Bitboard pawnsNotOn7 = pos.pieces(Us, PAWN) & ~TRank7BB;

    // Single and double pawn pushes, no promotions
    if constexpr (Type != CAPTURES)
    {
        Bitboard b1 = shift<Up>(pawnsNotOn7) & emptySquares;
        Bitboard b2 = shift<Up>(b1 & TRank3BB) & emptySquares;

        if constexpr (Type == EVASIONS)  // Consider only blocking squares
        {
            b1 &= target;
            b2 &= target;
        }

        while (b1)
        {
            Square to   = pop_lsb(b1);
            *moveList++ = Move(to - Up, to);
        }

        while (b2)
        {
            Square to   = pop_lsb(b2);
            *moveList++ = Move(to - Up - Up, to);
        }
    }

    // Promotions and underpromotions
    if (pawnsOn7)
    {
        Bitboard b1 = shift<UpRight>(pawnsOn7) & enemies;
        Bitboard b2 = shift<UpLeft>(pawnsOn7) & enemies;
        Bitboard b3 = shift<Up>(pawnsOn7) & emptySquares;

        if constexpr (Type == EVASIONS)
            b3 &= target;

        while (b1)
            moveList = make_promotions<Type, UpRight, true>(moveList, pop_lsb(b1));

        while (b2)
            moveList = make_promotions<Type, UpLeft, true>(moveList, pop_lsb(b2));

        while (b3)
            moveList = make_promotions<Type, Up, false>(moveList, pop_lsb(b3));
    }

    // Standard and en passant captures
    if constexpr (Type == CAPTURES || Type == EVASIONS || Type == NON_EVASIONS)
    {
        Bitboard b1 = shift<UpRight>(pawnsNotOn7) & enemies;
        Bitboard b2 = shift<UpLeft>(pawnsNotOn7) & enemies;

        while (b1)
        {
            Square to   = pop_lsb(b1);
            *moveList++ = Move(to - UpRight, to);
        }

        while (b2)
        {
            Square to   = pop_lsb(b2);
            *moveList++ = Move(to - UpLeft, to);
        }

        if (pos.ep_square() != SQ_NONE)
        {
            assert(rank_of(pos.ep_square()) == relative_rank(Us, RANK_6));

            // An en passant capture cannot resolve a discovered check
            if (Type == EVASIONS && (target & (pos.ep_square() + Up)))
                return moveList;

            b1 = pawnsNotOn7 & pawn_attacks_bb(Them, pos.ep_square());

            assert(b1);

            while (b1)
                *moveList++ = Move::make<EN_PASSANT>(pop_lsb(b1), pos.ep_square());
        }
    }

    return moveList;
}

template<Color Us, PieceType Pt>
ExtMove* generate_moves(const Position& pos, ExtMove* moveList, Bitboard target) {

    static_assert(Pt != KING && Pt != PAWN, "Unsupported piece type in generate_moves()");

    Bitboard bb = pos.pieces(Us, Pt);

    while (bb)
    {
        Square   from = pop_lsb(bb);
        Bitboard b    = attacks_bb<Pt>(from, pos.pieces()) & target;

        while (b)
            *moveList++ = Move(from, pop_lsb(b));
    }

    return moveList;
}

template<Color Us, GenType Type>
ExtMove* generate_all(const Position& pos, ExtMove* moveList) {

    static_assert(Type != LEGAL, "Unsupported type in generate_all()");

    const Square ksq = pos.square<KING>(Us);
    Bitboard     target;

    // Skip generating non-king moves when in double check
    if (Type != EVASIONS || !more_than_one(pos.checkers()))
    {
        target = Type == EVASIONS     ? between_bb(ksq, lsb(pos.checkers()))
               : Type == NON_EVASIONS ? ~pos.pieces(Us)
               : Type == CAPTURES     ? pos.pieces(~Us)
                                      : ~pos.pieces();  // QUIETS

        moveList = generate_pawn_moves<Us, Type>(pos, moveList, target);
        moveList = generate_moves<Us, KNIGHT>(pos, moveList, target);
        moveList = generate_moves<Us, BISHOP>(pos, moveList, target);
        moveList = generate_moves<Us, ROOK>(pos, moveList, target);
        moveList = generate_moves<Us, QUEEN>(pos, moveList, target);
    }

    Bitboard b = attacks_bb<KING>(ksq) & (Type == EVASIONS ? ~pos.pieces(Us) : target);

    while (b)
        *moveList++ = Move(ksq, pop_lsb(b));

    if ((Type == QUIETS || Type == NON_EVASIONS) && pos.can_castle(Us & ANY_CASTLING))
        for (CastlingRights cr : {Us & KING_SIDE, Us & QUEEN_SIDE})
            if (!pos.castling_impeded(cr) && pos.can_castle(cr))
                *moveList++ = Move::make<CASTLING>(ksq, pos.castling_rook_square(cr));

    return moveList;
}
```

```
}  // namespace

// <CAPTURES>     Generates all pseudo-legal captures plus queen promotions
// <QUIETS>       Generates all pseudo-legal non-captures and underpromotions
// <EVASIONS>     Generates all pseudo-legal check evasions
// <NON_EVASIONS> Generates all pseudo-legal captures and non-captures
//
// Returns a pointer to the end of the move list.
template<GenType Type>
ExtMove* generate(const Position& pos, ExtMove* moveList) {

    static_assert(Type != LEGAL, "Unsupported type in generate()");
    assert((Type == EVASIONS) == bool(pos.checkers()));

    Color us = pos.side_to_move();

    return us == WHITE ? generate_all<WHITE, Type>(pos, moveList)
                       : generate_all<BLACK, Type>(pos, moveList);
}

// Explicit template instantiations
template ExtMove* generate<CAPTURES>(const Position&, ExtMove*);
template ExtMove* generate<QUIETS>(const Position&, ExtMove*);
template ExtMove* generate<EVASIONS>(const Position&, ExtMove*);
template ExtMove* generate<NON_EVASIONS>(const Position&, ExtMove*);

// generate<LEGAL> generates all the legal moves in the given position

template<>
ExtMove* generate<LEGAL>(const Position& pos, ExtMove* moveList) {

    Color    us     = pos.side_to_move();
    Bitboard pinned = pos.blockers_for_king(us) & pos.pieces(us);
    Square   ksq    = pos.square<KING>(us);
    ExtMove* cur    = moveList;

    moveList =
      pos.checkers() ? generate<EVASIONS>(pos, moveList) : generate<NON_EVASIONS>(pos, moveList);
    while (cur != moveList)
        if (((pinned & cur->from_sq()) || cur->from_sq() == ksq || cur->type_of() == EN_PASSANT)
            && !pos.legal(*cur))
            *cur = *(--moveList);
        else
            ++cur;

    return moveList;
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

namespace {

enum Stages {
    // generate main search moves
    MAIN_TT,
    CAPTURE_INIT,
    GOOD_CAPTURE,
    QUIET_INIT,
    GOOD_QUIET,
    BAD_CAPTURE,
    BAD_QUIET,

    // generate evasion moves
    EVASION_TT,
    EVASION_INIT,
    EVASION,

    // generate probcut moves
    PROBCUT_TT,
    PROBCUT_INIT,
    PROBCUT,

    // generate qsearch moves
    QSEARCH_TT,
    QCAPTURE_INIT,
    QCAPTURE
};

// Sort moves in descending order up to and including a given limit.
// The order of moves smaller than the limit is left unspecified.
void partial_insertion_sort(ExtMove* begin, ExtMove* end, int limit) {

    for (ExtMove *sortedEnd = begin, *p = begin + 1; p < end; ++p)
        if (p->value >= limit)
        {
            ExtMove tmp = *p, *q;
            *p          = *++sortedEnd;
            for (q = sortedEnd; q != begin && *(q - 1) < tmp; --q)
                *q = *(q - 1);
            *q = tmp;
        }
}

}  // namespace

// Constructors of the MovePicker class. As arguments, we pass information
// to decide which class of moves to emit, to help sorting the (presumably)
// good moves first, and how important move ordering is at the current node.

// MovePicker constructor for the main search and for the quiescence search
MovePicker::MovePicker(const Position&              p,
                       Move                         ttm,
                       Depth                        d,
                       const ButterflyHistory*      mh,
                       const CapturePieceToHistory* cph,
                       const PieceToHistory**       ch,
                       const PawnHistory*           ph) :
    pos(p),
    mainHistory(mh),
    captureHistory(cph),
    continuationHistory(ch),
    pawnHistory(ph),
    ttMove(ttm),
    depth(d) {

    if (pos.checkers())
        stage = EVASION_TT + !(ttm && pos.pseudo_legal(ttm));

    else
        stage = (depth > 0 ? MAIN_TT : QSEARCH_TT) + !(ttm && pos.pseudo_legal(ttm));
}

// MovePicker constructor for ProbCut: we generate captures with Static Exchange
// Evaluation (SEE) greater than or equal to the given threshold.
MovePicker::MovePicker(const Position& p, Move ttm, int th, const CapturePieceToHistory* cph) :
    pos(p),
    captureHistory(cph),
    ttMove(ttm),
    threshold(th) {
    assert(!pos.checkers());

    stage = PROBCUT_TT
          + !(ttm && pos.capture_stage(ttm) && pos.pseudo_legal(ttm) && pos.see_ge(ttm, threshold));
}

// Assigns a numerical value to each move in a list, used for sorting.
// Captures are ordered by Most Valuable Victim (MVV), preferring captures
// with a good history. Quiets moves are ordered using the history tables.
template<GenType Type>
void MovePicker::score() {

    static_assert(Type == CAPTURES || Type == QUIETS || Type == EVASIONS, "Wrong type");

    [[maybe_unused]] Bitboard threatenedByPawn, threatenedByMinor, threatenedByRook,
      threatenedPieces;
    if constexpr (Type == QUIETS)
    {
        Color us = pos.side_to_move();

        threatenedByPawn = pos.attacks_by<PAWN>(~us);
        threatenedByMinor =
          pos.attacks_by<KNIGHT>(~us) | pos.attacks_by<BISHOP>(~us) | threatenedByPawn;
        threatenedByRook = pos.attacks_by<ROOK>(~us) | threatenedByMinor;

        // Pieces threatened by pieces of lesser material value
        threatenedPieces = (pos.pieces(us, QUEEN) & threatenedByRook)
                         | (pos.pieces(us, ROOK) & threatenedByMinor)
                         | (pos.pieces(us, KNIGHT, BISHOP) & threatenedByPawn);
    }

    for (auto& m : *this)
        if constexpr (Type == CAPTURES)
            m.value =
              7 * int(PieceValue[pos.piece_on(m.to_sq())])
              + (*captureHistory)[pos.moved_piece(m)][m.to_sq()][type_of(pos.piece_on(m.to_sq()))];

        else if constexpr (Type == QUIETS)
        {
            Piece     pc   = pos.moved_piece(m);
            PieceType pt   = type_of(pc);
            Square    from = m.from_sq();
            Square    to   = m.to_sq();

            // histories
            m.value = (*mainHistory)[pos.side_to_move()][m.from_to()];
            m.value += 2 * (*pawnHistory)[pawn_structure_index(pos)][pc][to];
            m.value += 2 * (*continuationHistory[0])[pc][to];
            m.value += (*continuationHistory[1])[pc][to];
            m.value += (*continuationHistory[2])[pc][to] / 3;
            m.value += (*continuationHistory[3])[pc][to];
            m.value += (*continuationHistory[5])[pc][to];

            // bonus for checks
            m.value += bool(pos.check_squares(pt) & to) * 16384;

            // bonus for escaping from capture
            m.value += threatenedPieces & from ? (pt == QUEEN && !(to & threatenedByRook)   ? 51700
                                                  : pt == ROOK && !(to & threatenedByMinor) ? 25600
                                                  : !(to & threatenedByPawn)                ? 14450
                                                                                            : 0)
                                               : 0;

            // malus for putting piece en prise
            m.value -= (pt == QUEEN  ? bool(to & threatenedByRook) * 49000
                        : pt == ROOK ? bool(to & threatenedByMinor) * 24335
                                     : bool(to & threatenedByPawn) * 14900);
        }

        else  // Type == EVASIONS
        {
            if (pos.capture_stage(m))
                m.value =
                  PieceValue[pos.piece_on(m.to_sq())] - type_of(pos.moved_piece(m)) + (1 << 28);
            else
                m.value = (*mainHistory)[pos.side_to_move()][m.from_to()]
                        + (*continuationHistory[0])[pos.moved_piece(m)][m.to_sq()]
                        + (*pawnHistory)[pawn_structure_index(pos)][pos.moved_piece(m)][m.to_sq()];
        }
}

// Returns the next move satisfying a predicate function.
// This never returns the TT move, as it was emitted before.
template<MovePicker::PickType T, typename Pred>
Move MovePicker::select(Pred filter) {

    while (cur < endMoves)
    {
        if constexpr (T == Best)
            std::swap(*cur, *std::max_element(cur, endMoves));

        if (*cur != ttMove && filter())
            return *cur++;

        cur++;
    }
    return Move::none();
}

// This is the most important method of the MovePicker class. We emit one
// new pseudo-legal move on every call until there are no more moves left,
// picking the move with the highest score from a list of generated moves.
Move MovePicker::next_move(bool skipQuiets) {

    auto quiet_threshold = [](Depth d) { return -3560 * d; };

top:
    switch (stage)
    {

    case MAIN_TT :
    case EVASION_TT :
    case QSEARCH_TT :
    case PROBCUT_TT :
        ++stage;
        return ttMove;

    case CAPTURE_INIT :
    case PROBCUT_INIT :
    case QCAPTURE_INIT :
        cur = endBadCaptures = moves;
        endMoves             = generate<CAPTURES>(pos, cur);

        score<CAPTURES>();
        partial_insertion_sort(cur, endMoves, std::numeric_limits<int>::min());
        ++stage;
        goto top;

    case GOOD_CAPTURE :
        if (select<Next>([&]() {
                // Move losing capture to endBadCaptures to be tried later
                return pos.see_ge(*cur, -cur->value / 18) ? true
                                                           : (*endBadCaptures++ = *cur, false);
            }))
            return *(cur - 1);

        ++stage;
        [[fallthrough]];

    case QUIET_INIT :
        if (!skipQuiets)
        {
            cur      = endBadCaptures;
            endMoves = beginBadQuiets = endBadQuiets = generate<QUIETS>(pos, cur);

            score<QUIETS>();
            partial_insertion_sort(cur, endMoves, quiet_threshold(depth));
        }

        ++stage;
        [[fallthrough]];

    case GOOD_QUIET :
        if (!skipQuiets && select<Next>([]() { return true; }))
        {
            if ((cur - 1)->value > -7998 || (cur - 1)->value <= quiet_threshold(depth))
                return *(cur - 1);

            // Remaining quiets are bad
            beginBadQuiets = cur - 1;
        }

        // Prepare the pointers to loop over the bad captures
        cur      = moves;
        endMoves = endBadCaptures;

        ++stage;
        [[fallthrough]];

    case BAD_CAPTURE :
        if (select<Next>([]() { return true; }))
            return *(cur - 1);

        // Prepare the pointers to loop over the bad quiets
        cur      = beginBadQuiets;
        endMoves = endBadQuiets;

        ++stage;
        [[fallthrough]];

    case BAD_QUIET :
        if (!skipQuiets)
            return select<Next>([]() { return true; });

        return Move::none();

    case EVASION_INIT :
        cur      = moves;
        endMoves = generate<EVASIONS>(pos, cur);

        score<EVASIONS>();
        ++stage;
        [[fallthrough]];

    case EVASION :
        return select<Best>([]() { return true; });

    case PROBCUT :
        return select<Next>([&]() { return pos.see_ge(*cur, threshold); });

    case QCAPTURE :
        return select<Next>([]() { return true; });
    }

    assert(false);
    return Move::none();  // Silence warning
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

namespace TB = Tablebases;

void syzygy_extend_pv(const OptionsMap&            options,
                      const Search::LimitsType&    limits,
                      Stockfish::Position&         pos,
                      Stockfish::Search::RootMove& rootMove,
                      Value&                       v);

using Eval::evaluate;
using namespace Search;

namespace {

// Futility margin
Value futility_margin(Depth d, bool noTtCutNode, bool improving, bool oppWorsening) {
    Value futilityMult       = 122 - 37 * noTtCutNode;
    Value improvingDeduction = improving * futilityMult * 2;
    Value worseningDeduction = oppWorsening * futilityMult / 3;

    return futilityMult * d - improvingDeduction - worseningDeduction;
}

constexpr int futility_move_count(bool improving, Depth depth) {
    return (3 + depth * depth) / (2 - improving);
}

// Add correctionHistory value to raw staticEval and guarantee evaluation
// does not hit the tablebase range.
Value to_corrected_static_eval(Value v, const Worker& w, const Position& pos) {
    auto cv = w.correctionHistory[pos.side_to_move()][pawn_structure_index<Correction>(pos)];
    v += 66 * cv / 512;
    return std::clamp(v, VALUE_TB_LOSS_IN_MAX_PLY + 1, VALUE_TB_WIN_IN_MAX_PLY - 1);
}

// History and stats update bonus, based on depth
int stat_bonus(Depth d) { return std::min(190 * d - 108, 1596); }

// History and stats update malus, based on depth
int stat_malus(Depth d) { return std::min(736 * d - 268, 2044); }

// Add a small random component to draw evaluations to avoid 3-fold blindness
Value value_draw(size_t nodes) { return VALUE_DRAW - 1 + Value(nodes & 0x2); }
Value value_to_tt(Value v, int ply);
Value value_from_tt(Value v, int ply, int r50c);
void  update_pv(Move* pv, Move move, const Move* childPv);
void  update_continuation_histories(Stack* ss, Piece pc, Square to, int bonus);
void  update_quiet_histories(
   const Position& pos, Stack* ss, Search::Worker& workerThread, Move move, int bonus);
void update_all_stats(const Position&      pos,
                      Stack*               ss,
                      Search::Worker&      workerThread,
                      Move                 bestMove,
                      Square               prevSq,
                      ValueList<Move, 32>& quietsSearched,
                      ValueList<Move, 32>& capturesSearched,
                      Depth                depth);

}  // namespace

Search::Worker::Worker(SharedState&                    sharedState,
                       std::unique_ptr<ISearchManager> sm,
                       size_t                          threadId,
                       NumaReplicatedAccessToken       token) :
    // Unpack the SharedState struct into member variables
    threadIdx(threadId),
    numaAccessToken(token),
    manager(std::move(sm)),
    options(sharedState.options),
    threads(sharedState.threads),
    tt(sharedState.tt),
    networks(sharedState.networks),
    refreshTable(networks[token]) {
    clear();
}

void Search::Worker::ensure_network_replicated() {
    // Access once to force lazy initialization.
    // We do this because we want to avoid initialization during search.
    (void) (networks[numaAccessToken]);
}

void Search::Worker::start_searching() {

    // Non-main threads go directly to iterative_deepening()
    if (!is_mainthread())
    {
        iterative_deepening();
        return;
    }

    main_manager()->tm.init(limits, rootPos.side_to_move(), rootPos.game_ply(), options,
                            main_manager()->originalTimeAdjust);
    tt.new_search();

    if (rootMoves.empty())
    {
        rootMoves.emplace_back(Move::none());
        main_manager()->updates.onUpdateNoMoves(
          {0, {rootPos.checkers() ? -VALUE_MATE : VALUE_DRAW, rootPos}});
    }
    else
    {
        threads.start_searching();  // start non-main threads
        iterative_deepening();      // main thread start searching
    }

    // When we reach the maximum depth, we can arrive here without a raise of
    // threads.stop. However, if we are pondering or in an infinite search,
    // the UCI protocol states that we shouldn't print the best move before the
    // GUI sends a "stop" or "ponderhit" command. We therefore simply wait here
    // until the GUI sends one of those commands.
    while (!threads.stop && (main_manager()->ponder || limits.infinite))
    {}  // Busy wait for a stop or a ponder reset

    // Stop the threads if not already stopped (also raise the stop if
    // "ponderhit" just reset threads.ponder)
    threads.stop = true;

    // Wait until all threads have finished
    threads.wait_for_search_finished();

    // When playing in 'nodes as time' mode, subtract the searched nodes from
    // the available ones before exiting.
    if (limits.npmsec)
        main_manager()->tm.advance_nodes_time(threads.nodes_searched()
                                              - limits.inc[rootPos.side_to_move()]);

    Worker* bestThread = this;
    Skill   skill =
      Skill(options["Skill Level"], options["UCI_LimitStrength"] ? int(options["UCI_Elo"]) : 0);

    if (int(options["MultiPV"]) == 1 && !limits.depth && !limits.mate && !skill.enabled()
        && rootMoves[0].pv[0] != Move::none())
        bestThread = threads.get_best_thread()->worker.get();

    main_manager()->bestPreviousScore        = bestThread->rootMoves[0].score;
    main_manager()->bestPreviousAverageScore = bestThread->rootMoves[0].averageScore;

    // Send again PV info if we have a new best thread
    if (bestThread != this)
        main_manager()->pv(*bestThread, threads, tt, bestThread->completedDepth);

    std::string ponder;

    if (bestThread->rootMoves[0].pv.size() > 1
        || bestThread->rootMoves[0].extract_ponder_from_tt(tt, rootPos))
        ponder = UCIEngine::move(bestThread->rootMoves[0].pv[1], rootPos.is_chess960());

    auto bestmove = UCIEngine::move(bestThread->rootMoves[0].pv[0], rootPos.is_chess960());
    main_manager()->updates.onBestmove(bestmove, ponder);
}

// Main iterative deepening loop. It calls search()
// repeatedly with increasing depth until the allocated thinking time has been
// consumed, the user stops the search, or the maximum search depth is reached.
void Search::Worker::iterative_deepening() {

    SearchManager* mainThread = (is_mainthread() ? main_manager() : nullptr);

    Move pv[MAX_PLY + 1];

    Depth lastBestMoveDepth = 0;
    Value lastBestScore     = -VALUE_INFINITE;
    auto  lastBestPV        = std::vector{Move::none()};

    Value  alpha, beta;
    Value  bestValue     = -VALUE_INFINITE;
    Color  us            = rootPos.side_to_move();
    double timeReduction = 1, totBestMoveChanges = 0;
    int    delta, iterIdx                        = 0;

    // Allocate stack with extra size to allow access from (ss - 7) to (ss + 2):
    // (ss - 7) is needed for update_continuation_histories(ss - 1) which accesses (ss - 6),
    // (ss + 2) is needed for initialization of cutOffCnt.
    Stack  stack[MAX_PLY + 10] = {};
    Stack* ss                  = stack + 7;

    for (int i = 7; i > 0; --i)
    {
        (ss - i)->continuationHistory =
          &this->continuationHistory[0][0][NO_PIECE][0];  // Use as a sentinel
        (ss - i)->staticEval = VALUE_NONE;
    }

    for (int i = 0; i <= MAX_PLY + 2; ++i)
        (ss + i)->ply = i;

    ss->pv = pv;

    if (mainThread)
    {
        if (mainThread->bestPreviousScore == VALUE_INFINITE)
            mainThread->iterValue.fill(VALUE_ZERO);
        else
            mainThread->iterValue.fill(mainThread->bestPreviousScore);
    }

    size_t multiPV = size_t(options["MultiPV"]);
    Skill skill(options["Skill Level"], options["UCI_LimitStrength"] ? int(options["UCI_Elo"]) : 0);

    // When playing with strength handicap enable MultiPV search that we will
    // use behind-the-scenes to retrieve a set of possible moves.
    if (skill.enabled())
        multiPV = std::max(multiPV, size_t(4));

    multiPV = std::min(multiPV, rootMoves.size());

    int searchAgainCounter = 0;

    // Iterative deepening loop until requested to stop or the target depth is reached
    while (++rootDepth < MAX_PLY && !threads.stop
           && !(limits.depth && mainThread && rootDepth > limits.depth))
    {
        // Age out PV variability metric
        if (mainThread)
            totBestMoveChanges /= 2;

        // Save the last iteration's scores before the first PV line is searched and
        // all the move scores except the (new) PV are set to -VALUE_INFINITE.
        for (RootMove& rm : rootMoves)
            rm.previousScore = rm.score;

        size_t pvFirst = 0;
        pvLast         = 0;

        if (!threads.increaseDepth)
            searchAgainCounter++;

        // MultiPV loop. We perform a full root search for each PV line
        for (pvIdx = 0; pvIdx < multiPV; ++pvIdx)
        {
            if (pvIdx == pvLast)
            {
                pvFirst = pvLast;
                for (pvLast++; pvLast < rootMoves.size(); pvLast++)
                    if (rootMoves[pvLast].tbRank != rootMoves[pvFirst].tbRank)
                        break;
            }

            // Reset UCI info selDepth for each depth and each PV line
            selDepth = 0;

            // Reset aspiration window starting size
            Value avg = rootMoves[pvIdx].averageScore;
            delta     = 5 + avg * avg / 13424;
            alpha     = std::max(avg - delta, -VALUE_INFINITE);
            beta      = std::min(avg + delta, VALUE_INFINITE);

            // Adjust optimism based on root move's averageScore (~4 Elo)
            optimism[us]  = 125 * avg / (std::abs(avg) + 89);
            optimism[~us] = -optimism[us];

            // Start with a small aspiration window and, in the case of a fail
            // high/low, re-search with a bigger window until we don't fail
            // high/low anymore.
            int failedHighCnt = 0;
            while (true)
            {
                // Adjust the effective depth searched, but ensure at least one
                // effective increment for every four searchAgain steps (see issue #2717).
                Depth adjustedDepth =
                  std::max(1, rootDepth - failedHighCnt - 3 * (searchAgainCounter + 1) / 4);
                rootDelta = beta - alpha;
                bestValue = search<Root>(rootPos, ss, alpha, beta, adjustedDepth, false);

                // Bring the best move to the front. It is critical that sorting
                // is done with a stable algorithm because all the values but the
                // first and eventually the new best one is set to -VALUE_INFINITE
                // and we want to keep the same order for all the moves except the
                // new PV that goes to the front. Note that in the case of MultiPV
                // search the already searched PV lines are preserved.
                std::stable_sort(rootMoves.begin() + pvIdx, rootMoves.begin() + pvLast);

                // If search has been stopped, we break immediately. Sorting is
                // safe because RootMoves is still valid, although it refers to
                // the previous iteration.
                if (threads.stop)
                    break;

                // When failing high/low give some update before a re-search. To avoid
                // excessive output that could hang GUIs like Fritz 19, only start
                // at nodes > 10M (rather than depth N, which can be reached quickly)
                if (mainThread && multiPV == 1 && (bestValue <= alpha || bestValue >= beta)
                    && nodes > 10000000)
                    main_manager()->pv(*this, threads, tt, rootDepth);

                // In case of failing low/high increase aspiration window and re-search,
                // otherwise exit the loop.
                if (bestValue <= alpha)
                {
                    beta  = (alpha + beta) / 2;
                    alpha = std::max(bestValue - delta, -VALUE_INFINITE);

                    failedHighCnt = 0;
                    if (mainThread)
                        mainThread->stopOnPonderhit = false;
                }
                else if (bestValue >= beta)
                {
                    beta = std::min(bestValue + delta, VALUE_INFINITE);
                    ++failedHighCnt;
                }
                else
                    break;

                delta += delta / 3;

                assert(alpha >= -VALUE_INFINITE && beta <= VALUE_INFINITE);
            }

            // Sort the PV lines searched so far and update the GUI
            std::stable_sort(rootMoves.begin() + pvFirst, rootMoves.begin() + pvIdx + 1);

            if (mainThread
                && (threads.stop || pvIdx + 1 == multiPV || nodes > 10000000)
                // A thread that aborted search can have mated-in/TB-loss PV and
                // score that cannot be trusted, i.e. it can be delayed or refuted
                // if we would have had time to fully search other root-moves. Thus
                // we suppress this output and below pick a proven score/PV for this
                // thread (from the previous iteration).
                && !(threads.abortedSearch && rootMoves[0].uciScore <= VALUE_TB_LOSS_IN_MAX_PLY))
                main_manager()->pv(*this, threads, tt, rootDepth);

            if (threads.stop)
                break;
        }

        if (!threads.stop)
            completedDepth = rootDepth;

        // We make sure not to pick an unproven mated-in score,
        // in case this thread prematurely stopped search (aborted-search).
        if (threads.abortedSearch && rootMoves[0].score != -VALUE_INFINITE
            && rootMoves[0].score <= VALUE_TB_LOSS_IN_MAX_PLY)
        {
            // Bring the last best move to the front for best thread selection.
            Utility::move_to_front(rootMoves, [&lastBestPV = std::as_const(lastBestPV)](
                                                const auto& rm) { return rm == lastBestPV[0]; });
            rootMoves[0].pv    = lastBestPV;
            rootMoves[0].score = rootMoves[0].uciScore = lastBestScore;
        }
        else if (rootMoves[0].pv[0] != lastBestPV[0])
        {
            lastBestPV        = rootMoves[0].pv;
            lastBestScore     = rootMoves[0].score;
            lastBestMoveDepth = rootDepth;
        }

        if (!mainThread)
            continue;

        // Have we found a "mate in x"?
        if (limits.mate && rootMoves[0].score == rootMoves[0].uciScore
            && ((rootMoves[0].score >= VALUE_MATE_IN_MAX_PLY
                 && VALUE_MATE - rootMoves[0].score <= 2 * limits.mate)
                || (rootMoves[0].score != -VALUE_INFINITE
                    && rootMoves[0].score <= VALUE_MATED_IN_MAX_PLY
                    && VALUE_MATE + rootMoves[0].score <= 2 * limits.mate)))
            threads.stop = true;

        // If the skill level is enabled and time is up, pick a sub-optimal best move
        if (skill.enabled() && skill.time_to_pick(rootDepth))
            skill.pick_best(rootMoves, multiPV);

        // Use part of the gained time from a previous stable move for the current move
        for (auto&& th : threads)
        {
            totBestMoveChanges += th->worker->bestMoveChanges;
            th->worker->bestMoveChanges = 0;
        }

        // Do we have time for the next iteration? Can we stop searching now?
        if (limits.use_time_management() && !threads.stop && !mainThread->stopOnPonderhit)
        {
            int nodesEffort = rootMoves[0].effort * 100 / std::max(size_t(1), size_t(nodes));

            double fallingEval = (1067 + 223 * (mainThread->bestPreviousAverageScore - bestValue)
                                  + 97 * (mainThread->iterValue[iterIdx] - bestValue))
                               / 10000.0;
            fallingEval = std::clamp(fallingEval, 0.580, 1.667);

            // If the bestMove is stable over several iterations, reduce time accordingly
            timeReduction    = lastBestMoveDepth + 8 < completedDepth ? 1.495 : 0.687;
            double reduction = (1.48 + mainThread->previousTimeReduction) / (2.17 * timeReduction);
            double bestMoveInstability = 1 + 1.88 * totBestMoveChanges / threads.size();
            double recapture           = limits.capSq == rootMoves[0].pv[0].to_sq() ? 0.955 : 1.005;

            double totalTime =
              mainThread->tm.optimum() * fallingEval * reduction * bestMoveInstability * recapture;

            // Cap used time in case of a single legal move for a better viewer experience
            if (rootMoves.size() == 1)
                totalTime = std::min(500.0, totalTime);

            auto elapsedTime = elapsed();

            if (completedDepth >= 10 && nodesEffort >= 97 && elapsedTime > totalTime * 0.739
                && !mainThread->ponder)
                threads.stop = true;

            // Stop the search if we have exceeded the totalTime
            if (elapsedTime > totalTime)
            {
                // If we are allowed to ponder do not stop the search now but
                // keep pondering until the GUI sends "ponderhit" or "stop".
                if (mainThread->ponder)
                    mainThread->stopOnPonderhit = true;
                else
                    threads.stop = true;
            }
            else
                threads.increaseDepth = mainThread->ponder || elapsedTime <= totalTime * 0.506;
        }

        mainThread->iterValue[iterIdx] = bestValue;
        iterIdx                        = (iterIdx + 1) & 3;
    }

    if (!mainThread)
        return;

    mainThread->previousTimeReduction = timeReduction;

    // If the skill level is enabled, swap the best PV line with the sub-optimal one
    if (skill.enabled())
        std::swap(rootMoves[0],
                  *std::find(rootMoves.begin(), rootMoves.end(),
                             skill.best ? skill.best : skill.pick_best(rootMoves, multiPV)));
}

// Reset histories, usually before a new game
void Search::Worker::clear() {
    mainHistory.fill(0);
    captureHistory.fill(-700);
    pawnHistory.fill(-1188);
    correctionHistory.fill(0);

    for (bool inCheck : {false, true})
        for (StatsType c : {NoCaptures, Captures})
            for (auto& to : continuationHistory[inCheck][c])
                for (auto& h : to)
                    h->fill(-658);

    for (size_t i = 1; i < reductions.size(); ++i)
        reductions[i] = int((18.62 + std::log(size_t(options["Threads"])) / 2) * std::log(i));

    refreshTable.clear(networks[numaAccessToken]);
}

// Main search function for both PV and non-PV nodes
template<NodeType nodeType>
Value Search::Worker::search(
  Position& pos, Stack* ss, Value alpha, Value beta, Depth depth, bool cutNode) {

    constexpr bool PvNode   = nodeType != NonPV;
    constexpr bool rootNode = nodeType == Root;
    const bool     allNode  = !(PvNode || cutNode);

    // Dive into quiescence search when the depth reaches zero
    if (depth <= 0)
        return qsearch < PvNode ? PV : NonPV > (pos, ss, alpha, beta);

    // Limit the depth if extensions made it too large
    depth = std::min(depth, MAX_PLY - 1);

    // Check if we have an upcoming move that draws by repetition
    if (!rootNode && alpha < VALUE_DRAW && pos.upcoming_repetition(ss->ply))
    {
        alpha = value_draw(this->nodes);
        if (alpha >= beta)
            return alpha;
    }

    assert(-VALUE_INFINITE <= alpha && alpha < beta && beta <= VALUE_INFINITE);
    assert(PvNode || (alpha == beta - 1));
    assert(0 < depth && depth < MAX_PLY);
    assert(!(PvNode && cutNode));

    Move      pv[MAX_PLY + 1];
    StateInfo st;
    ASSERT_ALIGNED(&st, Eval::NNUE::CacheLineSize);

    Key   posKey;
    Move  move, excludedMove, bestMove;
    Depth extension, newDepth;
    Value bestValue, value, eval, maxValue, probCutBeta;
    bool  givesCheck, improving, priorCapture, opponentWorsening;
    bool  capture, ttCapture;
    Piece movedPiece;

    ValueList<Move, 32> capturesSearched;
    ValueList<Move, 32> quietsSearched;

    // Step 1. Initialize node
    Worker* thisThread = this;
    ss->inCheck        = pos.checkers();
    priorCapture       = pos.captured_piece();
    Color us           = pos.side_to_move();
    ss->moveCount      = 0;
    bestValue          = -VALUE_INFINITE;
    maxValue           = VALUE_INFINITE;

    // Check for the available remaining time
    if (is_mainthread())
        main_manager()->check_time(*thisThread);

    // Used to send selDepth info to GUI (selDepth counts from 1, ply from 0)
    if (PvNode && thisThread->selDepth < ss->ply + 1)
        thisThread->selDepth = ss->ply + 1;

    if (!rootNode)
    {
        // Step 2. Check for aborted search and immediate draw
        if (threads.stop.load(std::memory_order_relaxed) || pos.is_draw(ss->ply)
            || ss->ply >= MAX_PLY)
            return (ss->ply >= MAX_PLY && !ss->inCheck)
                   ? evaluate(networks[numaAccessToken], pos, refreshTable,
                              thisThread->optimism[us])
                   : value_draw(thisThread->nodes);

        // Step 3. Mate distance pruning. Even if we mate at the next move our score
        // would be at best mate_in(ss->ply + 1), but if alpha is already bigger because
        // a shorter mate was found upward in the tree then there is no need to search
        // because we will never beat the current alpha. Same logic but with reversed
        // signs apply also in the opposite condition of being mated instead of giving
        // mate. In this case, return a fail-high score.
        alpha = std::max(mated_in(ss->ply), alpha);
        beta  = std::min(mate_in(ss->ply + 1), beta);
        if (alpha >= beta)
            return alpha;
    }

    assert(0 <= ss->ply && ss->ply < MAX_PLY);

    bestMove            = Move::none();
    (ss + 2)->cutoffCnt = 0;
    Square prevSq = ((ss - 1)->currentMove).is_ok() ? ((ss - 1)->currentMove).to_sq() : SQ_NONE;
    ss->statScore = 0;

    // Step 4. Transposition table lookup
    excludedMove                   = ss->excludedMove;
    posKey                         = pos.key();
    auto [ttHit, ttData, ttWriter] = tt.probe(posKey);
    // Need further processing of the saved data
    ss->ttHit    = ttHit;
    ttData.move  = rootNode ? thisThread->rootMoves[thisThread->pvIdx].pv[0]
                 : ttHit    ? ttData.move
                            : Move::none();
    ttData.value = ttHit ? value_from_tt(ttData.value, ss->ply, pos.rule50_count()) : VALUE_NONE;
    ss->ttPv     = excludedMove ? ss->ttPv : PvNode || (ttHit && ttData.is_pv);
    ttCapture    = ttData.move && pos.capture_stage(ttData.move);

    // At this point, if excluded, skip straight to step 6, static eval. However,
    // to save indentation, we list the condition in all code between here and there.

    // At non-PV nodes we check for an early TT cutoff
    if (!PvNode && !excludedMove && ttData.depth > depth - (ttData.value <= beta)
        && ttData.value != VALUE_NONE  // Can happen when !ttHit or when access race in probe()
        && (ttData.bound & (ttData.value >= beta ? BOUND_LOWER : BOUND_UPPER)))
    {
        // If ttMove is quiet, update move sorting heuristics on TT hit (~2 Elo)
        if (ttData.move && ttData.value >= beta)
        {
            // Bonus for a quiet ttMove that fails high (~2 Elo)
            if (!ttCapture)
                update_quiet_histories(pos, ss, *this, ttData.move, stat_bonus(depth));

            // Extra penalty for early quiet moves of
            // the previous ply (~1 Elo on STC, ~2 Elo on LTC)
            if (prevSq != SQ_NONE && (ss - 1)->moveCount <= 2 && !priorCapture)
                update_continuation_histories(ss - 1, pos.piece_on(prevSq), prevSq,
                                             -stat_malus(depth + 1));
        }

        // Partial workaround for the graph history interaction problem
        // For high rule50 counts don't produce transposition table cutoffs.
        if (pos.rule50_count() < 90)
            return ttData.value;
    }

    // Step 5. Tablebases probe
    if (!rootNode && !excludedMove && tbConfig.cardinality)
    {
        int piecesCount = pos.count<ALL_PIECES>();

        if (piecesCount <= tbConfig.cardinality
            && (piecesCount < tbConfig.cardinality || depth >= tbConfig.probeDepth)
            && pos.rule50_count() == 0 && !pos.can_castle(ANY_CASTLING))
        {
            TB::ProbeState err;
            TB::WDLScore   wdl = Tablebases::probe_wdl(pos, &err);

            // Force check of time on the next occasion
            if (is_mainthread())
                main_manager()->callsCnt = 0;

            if (err != TB::ProbeState::FAIL)
            {
                thisThread->tbHits.fetch_add(1, std::memory_order_relaxed);

                int drawScore = tbConfig.useRule50 ? 1 : 0;

                Value tbValue = VALUE_TB - ss->ply;

                // Use the range VALUE_TB to VALUE_TB_WIN_IN_MAX_PLY to score
                value = wdl < -drawScore ? -tbValue
                      : wdl > drawScore  ? tbValue
                                         : VALUE_DRAW + 2 * wdl * drawScore;

                Bound b = wdl < -drawScore ? BOUND_UPPER
                        : wdl > drawScore  ? BOUND_LOWER
                                           : BOUND_EXACT;

                if (b == BOUND_EXACT || (b == BOUND_LOWER ? value >= beta : value <= alpha))
                {
                    ttWriter.write(posKey, value_to_tt(value, ss->ply), ss->ttPv, b,
                                   std::min(MAX_PLY - 1, depth + 6), Move::none(), VALUE_NONE,
                                   tt.generation());

                    return value;
                }

                if (PvNode)
                {
                    if (b == BOUND_LOWER)
                        bestValue = value, alpha = std::max(alpha, bestValue);
                    else
                        maxValue = value;
                }
            }
        }
    }

    // Step 6. Static evaluation of the position
    Value unadjustedStaticEval = VALUE_NONE;
    if (ss->inCheck)
    {
        // Skip early pruning when in check
        ss->staticEval = eval = (ss - 2)->staticEval;
        improving             = false;
        goto moves_loop;
    }
    else if (excludedMove)
    {
        // Providing the hint that this node's accumulator will be used often
        // brings significant Elo gain (~13 Elo).
        Eval::NNUE::hint_common_parent_position(pos, networks[numaAccessToken], refreshTable);
        unadjustedStaticEval = eval = ss->staticEval;
    }
    else if (ss->ttHit)
    {
        // Never assume anything about values stored in TT
        unadjustedStaticEval = ttData.eval;
        if (unadjustedStaticEval == VALUE_NONE)
            unadjustedStaticEval =
              evaluate(networks[numaAccessToken], pos, refreshTable, thisThread->optimism[us]);
        else if (PvNode)
            Eval::NNUE::hint_common_parent_position(pos, networks[numaAccessToken], refreshTable);

        ss->staticEval = eval = to_corrected_static_eval(unadjustedStaticEval, *thisThread, pos);

        // ttValue can be used as a better position evaluation (~7 Elo)
        if (ttData.value != VALUE_NONE
            && (ttData.bound & (ttData.value > eval ? BOUND_LOWER : BOUND_UPPER)))
            eval = ttData.value;
    }
    else
    {
        unadjustedStaticEval =
          evaluate(networks[numaAccessToken], pos, refreshTable, thisThread->optimism[us]);
        ss->staticEval = eval = to_corrected_static_eval(unadjustedStaticEval, *thisThread, pos);

        // Static evaluation is saved as it was before adjustment by correction history
        ttWriter.write(posKey, VALUE_NONE, ss->ttPv, BOUND_NONE, DEPTH_UNSEARCHED, Move::none(),
                       unadjustedStaticEval, tt.generation());
    }

    // Use static evaluation difference to improve quiet move ordering (~9 Elo)
    if (((ss - 1)->currentMove).is_ok() && !(ss - 1)->inCheck && !priorCapture)
    {
        int bonus = std::clamp(-10 * int((ss - 1)->staticEval + ss->staticEval), -1664, 1471) + 752;
        thisThread->mainHistory[~us][((ss - 1)->currentMove).from_to()] << bonus;
        if (type_of(pos.piece_on(prevSq)) != PAWN && ((ss - 1)->currentMove).type_of() != PROMOTION)
            thisThread->pawnHistory[pawn_structure_index(pos)][pos.piece_on(prevSq)][prevSq]
              << bonus / 2;
    }

    // Set up the improving flag, which is true if current static evaluation is
    // bigger than the previous static evaluation at our turn (if we were in
    // check at our previous move we go back until we weren't in check) and is
    // false otherwise. The improving flag is used in various pruning heuristics.
    improving = ss->staticEval > (ss - 2)->staticEval;

    opponentWorsening = ss->staticEval + (ss - 1)->staticEval > 2;

    // Step 7. Razoring (~1 Elo)
    // If eval is really low, check with qsearch if we can exceed alpha. If the
    // search suggests we cannot exceed alpha, return a speculative fail low.
    if (eval < alpha - 494 - 290 * depth * depth)
    {
        value = qsearch<NonPV>(pos, ss, alpha - 1, alpha);
        if (value < alpha && std::abs(value) < VALUE_TB_WIN_IN_MAX_PLY)
            return value;
    }

    // Step 8. Futility pruning: child node (~40 Elo)
    // The depth condition is important for mate finding.
    if (!ss->ttPv && depth < 13
        && eval - futility_margin(depth, cutNode && !ss->ttHit, improving, opponentWorsening)
               - (ss - 1)->statScore / 260
             >= beta
        && eval >= beta && (!ttData.move || ttCapture) && beta > VALUE_TB_LOSS_IN_MAX_PLY
        && eval < VALUE_TB_WIN_IN_MAX_PLY)
        return beta + (eval - beta) / 3;

    // Step 9. Null move search with verification search (~35 Elo)
    if (cutNode && (ss - 1)->currentMove != Move::null() && (ss - 1)->statScore < 14389
        && eval >= beta && ss->staticEval >= beta - 21 * depth + 390 && !excludedMove
        && pos.non_pawn_material(us) && ss->ply >= thisThread->nmpMinPly
        && beta > VALUE_TB_LOSS_IN_MAX_PLY)
    {
        assert(eval - beta >= 0);

        // Null move dynamic reduction based on depth and eval
        Depth R = std::min(int(eval - beta) / 202, 6) + depth / 3 + 5;

        ss->currentMove         = Move::null();
        ss->continuationHistory = &thisThread->continuationHistory[0][0][NO_PIECE][0];

        pos.do_null_move(st, tt);

        Value nullValue = -search<NonPV>(pos, ss + 1, -beta, -beta + 1, depth - R, false);

        pos.undo_null_move();

        // Do not return unproven mate or TB scores
        if (nullValue >= beta && nullValue < VALUE_TB_WIN_IN_MAX_PLY)
        {
            if (thisThread->nmpMinPly || depth < 16)
                return nullValue;

            assert(!thisThread->nmpMinPly);  // Recursive verification is not allowed

            // Do verification search at high depths, with null move pruning disabled
            // until ply exceeds nmpMinPly.
            thisThread->nmpMinPly = ss->ply + 3 * (depth - R) / 4;

            Value v = search<NonPV>(pos, ss, beta - 1, beta, depth - R, false);

            thisThread->nmpMinPly = 0;

            if (v >= beta)
                return nullValue;
        }
    }

    // Step 10. Internal iterative reductions (~9 Elo)
    // For PV nodes without a ttMove, we decrease depth.
    if (PvNode && !ttData.move)
        depth -= 3;

    // Use qsearch if depth <= 0
    if (depth <= 0)
        return qsearch<PV>(pos, ss, alpha, beta);

    // For cutNodes, if depth is high enough, decrease depth by 2 if there is no ttMove,
    // or by 1 if there is a ttMove with an upper bound.
    if (cutNode && depth >= 7 && (!ttData.move || ttData.bound == BOUND_UPPER))
        depth -= 1 + !ttData.move;

    // Step 11. ProbCut (~10 Elo)
    // If we have a good enough capture (or queen promotion) and a reduced search
    // returns a value much above beta, we can (almost) safely prune the previous move.
    probCutBeta = beta + 184 - 53 * improving;
    if (!PvNode && depth > 3
        && std::abs(beta) < VALUE_TB_WIN_IN_MAX_PLY
        // If value from transposition table is lower than probCutBeta, don't attempt
        // probCut there and in further interactions with transposition table cutoff
        // depth is set to depth - 3 because probCut search has depth set to depth - 4
        // but we also do a move before it. So effective depth is equal to depth - 3.
        && !(ttData.depth >= depth - 3 && ttData.value != VALUE_NONE && ttData.value < probCutBeta))
    {
        assert(probCutBeta < VALUE_INFINITE && probCutBeta > beta);

        MovePicker mp(pos, ttData.move, probCutBeta - ss->staticEval, &thisThread->captureHistory);
        Piece      captured;

        while ((move = mp.next_move()) != Move::none())
        {
            assert(move.is_ok());

            if (move == excludedMove)
                continue;

            if (!pos.legal(move))
                continue;

            assert(pos.capture_stage(move));

            movedPiece = pos.moved_piece(move);
            captured   = pos.piece_on(move.to_sq());

            // Prefetch the TT entry for the resulting position
            prefetch(tt.first_entry(pos.key_after(move)));

            ss->currentMove = move;
            ss->continuationHistory =
              &this->continuationHistory[ss->inCheck][true][pos.moved_piece(move)][move.to_sq()];

            thisThread->nodes.fetch_add(1, std::memory_order_relaxed);
            pos.do_move(move, st);

            // Perform a preliminary qsearch to verify that the move holds
            value = -qsearch<NonPV>(pos, ss + 1, -probCutBeta, -probCutBeta + 1);

            // If the qsearch held, perform the regular search
            if (value >= probCutBeta)
                value =
                  -search<NonPV>(pos, ss + 1, -probCutBeta, -probCutBeta + 1, depth - 4, !cutNode);

            pos.undo_move(move);

            if (value >= probCutBeta)
            {
                thisThread->captureHistory[movedPiece][move.to_sq()][type_of(captured)]
                  << stat_bonus(depth - 2);

                // Save ProbCut data into transposition table
                ttWriter.write(posKey, value_to_tt(value, ss->ply), ss->ttPv, BOUND_LOWER,
                               depth - 3, move, unadjustedStaticEval, tt.generation());
                return std::abs(value) < VALUE_TB_WIN_IN_MAX_PLY ? value - (probCutBeta - beta)
                                                                 : value;
            }
        }

        Eval::NNUE::hint_common_parent_position(pos, networks[numaAccessToken], refreshTable);
    }

moves_loop:  // When in check, search starts here

    // Step 12. A small Probcut idea (~4 Elo)
    probCutBeta = beta + 390;
    if ((ttData.bound & BOUND_LOWER) && ttData.depth >= depth - 4 && ttData.value >= probCutBeta
        && std::abs(beta) < VALUE_TB_WIN_IN_MAX_PLY
        && std::abs(ttData.value) < VALUE_TB_WIN_IN_MAX_PLY)
        return probCutBeta;

    const PieceToHistory* contHist[] = {(ss - 1)->continuationHistory,
                                        (ss - 2)->continuationHistory,
                                        (ss - 3)->continuationHistory,
                                        (ss - 4)->continuationHistory,
                                        nullptr,
                                        (ss - 6)->continuationHistory};

    MovePicker mp(pos, ttData.move, depth, &thisThread->mainHistory, &thisThread->captureHistory,
                  contHist, &thisThread->pawnHistory);

    value = bestValue;

    int  moveCount        = 0;
    bool moveCountPruning = false;

    // Step 13. Loop through all pseudo-legal moves until no moves remain
    // or a beta cutoff occurs.
    while ((move = mp.next_move(moveCountPruning)) != Move::none())
    {
        assert(move.is_ok());

        if (move == excludedMove)
            continue;

        // Check for legality
        if (!pos.legal(move))
            continue;

        // At root obey the "searchmoves" option and skip moves not listed in Root
        // Move List. In MultiPV mode we also skip PV moves that have been already
        // searched and those of lower "TB rank" if we are in a TB root position.
        if (rootNode
            && !std::count(thisThread->rootMoves.begin() + thisThread->pvIdx,
                           thisThread->rootMoves.begin() + thisThread->pvLast, move))
            continue;

        ss->moveCount = ++moveCount;

        if (rootNode && is_mainthread() && nodes > 10000000)
        {
            main_manager()->updates.onIter(
              {depth, UCIEngine::move(move, pos.is_chess960()), moveCount + thisThread->pvIdx});
        }
        if (PvNode)
            (ss + 1)->pv = nullptr;

        extension  = 0;
        capture    = pos.capture_stage(move);
        movedPiece = pos.moved_piece(move);
        givesCheck = pos.gives_check(move);

        // Calculate new depth for this move
        newDepth = depth - 1;

        int delta = beta - alpha;

        Depth r = reduction(improving, depth, moveCount, delta);

        // Step 14. Pruning at shallow depth (~120 Elo).
        // Depth conditions are important for mate finding.
        if (!rootNode && pos.non_pawn_material(us) && bestValue > VALUE_TB_LOSS_IN_MAX_PLY)
        {
            // Skip quiet moves if movecount exceeds our FutilityMoveCount threshold (~8 Elo)
            moveCountPruning = moveCount >= futility_move_count(improving, depth);

            // Reduced depth of the next LMR search
            int lmrDepth = newDepth - r;

            if (capture || givesCheck)
            {
                Piece capturedPiece = pos.piece_on(move.to_sq());
                int   captHist =
                  thisThread->captureHistory[movedPiece][move.to_sq()][type_of(capturedPiece)];

                // Futility pruning for captures (~2 Elo)
                if (!givesCheck && lmrDepth < 7 && !ss->inCheck)
                {
                    Value futilityValue = ss->staticEval + 285 + 251 * lmrDepth
                                        + PieceValue[capturedPiece] + captHist / 7;
                    if (futilityValue <= alpha)
                        continue;
                }

                // SEE based pruning for captures and checks (~11 Elo)
                int seeHist = std::clamp(captHist / 32, -182 * depth, 166 * depth);
                if (!pos.see_ge(move, -168 * depth - seeHist))
                    continue;
            }
            else
            {
                int history =
                  (*contHist[0])[movedPiece][move.to_sq()]
                  + (*contHist[1])[movedPiece][move.to_sq()]
                  + thisThread->pawnHistory[pawn_structure_index(pos)][movedPiece][move.to_sq()];

                // Continuation history based pruning (~2 Elo)
                if (history < -4165 * depth)
                    continue;

                history += 2 * thisThread->mainHistory[us][move.from_to()];

                lmrDepth += history / 3853;

                Value futilityValue =
                  ss->staticEval + (bestValue < ss->staticEval - 51 ? 143 : 52) + 135 * lmrDepth;

                // Futility pruning: parent node (~13 Elo)
                if (!ss->inCheck && lmrDepth < 12 && futilityValue <= alpha)
                {
                    if (bestValue <= futilityValue && std::abs(bestValue) < VALUE_TB_WIN_IN_MAX_PLY
                        && futilityValue < VALUE_TB_WIN_IN_MAX_PLY)
                        bestValue = futilityValue;
                    continue;
                }

                lmrDepth = std::max(lmrDepth, 0);

                // Prune moves with negative SEE (~4 Elo)
                if (!pos.see_ge(move, -24 * lmrDepth * lmrDepth))
                    continue;
            }
        }

        // Step 15. Extensions (~100 Elo)
        // We take care to not overdo to avoid search getting stuck.
        if (ss->ply < thisThread->rootDepth * 2)
        {
            // Singular extension search (~76 Elo, ~170 nElo). If all moves but one
            // fail low on a search of (alpha-s, beta-s), and just one fails high on
            // (alpha, beta), then that move is singular and should be extended. To
            // verify this we do a reduced search on the position excluding the ttMove
            // and if the result is lower than ttValue minus a margin, then we will
            //  extend the ttMove. Recursive singular search is avoided.

            // Note: the depth margin and singularBeta margin are known for having
            // non-linear scaling. Their values are optimized to time controls of
            // 180+1.8 and longer so changing them requires tests at these types of
            // time controls. Generally, higher singularBeta (i.e closer to ttValue)
            // and lower extension margins scale well.

            if (!rootNode && move == ttData.move && !excludedMove
                && depth >= 4 - (thisThread->completedDepth > 36) + ss->ttPv
                && std::abs(ttData.value) < VALUE_TB_WIN_IN_MAX_PLY && (ttData.bound & BOUND_LOWER)
                && ttData.depth >= depth - 3)
            {
                Value singularBeta  = ttData.value - (54 + 76 * (ss->ttPv && !PvNode)) * depth / 64;
                Depth singularDepth = newDepth / 2;

                ss->excludedMove = move;
                value =
                  search<NonPV>(pos, ss, singularBeta - 1, singularBeta, singularDepth, cutNode);
                ss->excludedMove = Move::none();

                if (value < singularBeta)
                {
                    int doubleMargin = 293 * PvNode - 195 * !ttCapture;
                    int tripleMargin = 107 + 259 * PvNode - 260 * !ttCapture + 98 * ss->ttPv;

                    extension = 1 + (value < singularBeta - doubleMargin)
                              + (value < singularBeta - tripleMargin);

                    depth += ((!PvNode) && (depth < 16));
                }

                // Multi-cut pruning
                // Our ttMove is assumed to fail high based on the bound of the TT entry,
                // and if after excluding the ttMove with a reduced search we fail high
                // over the original beta, we assume this expected cut-node is not
                // singular (multiple moves fail high), and we can prune the whole
                // subtree by returning a softbound.
                else if (value >= beta && std::abs(value) < VALUE_TB_WIN_IN_MAX_PLY)
                    return value;

                // Negative extensions
                // If other moves failed high over (ttValue - margin) without the
                // ttMove on a reduced search, but we cannot do multi-cut because
                // (ttValue - margin) is lower than the original beta, we do not know
                // if the ttMove is singular or can do a multi-cut, so we reduce the
                // ttMove in favor of other moves based on some conditions:

                // If the ttMove is assumed to fail high over current beta (~7 Elo)
                else if (ttData.value >= beta)
                    extension = -3;

                // If we are on a cutNode but the ttMove is not assumed to fail high
                // over current beta (~1 Elo)
                else if (cutNode)
                    extension = -2;
            }

            // Extension for capturing the previous moved piece (~1 Elo at LTC)
            else if (PvNode && move.to_sq() == prevSq
                     && thisThread->captureHistory[movedPiece][move.to_sq()]
                                                  [type_of(pos.piece_on(move.to_sq()))]
                          > 3994)
                extension = 1;
        }

        // Add extension to new depth
        newDepth += extension;

        // Speculative prefetch as early as possible
        prefetch(tt.first_entry(pos.key_after(move)));

        // Update the current move (this must be done after singular extension search)
        ss->currentMove = move;
        ss->continuationHistory =
          &thisThread->continuationHistory[ss->inCheck][capture][movedPiece][move.to_sq()];

        uint64_t nodeCount = rootNode ? uint64_t(nodes) : 0;

        // Step 16. Make the move
        thisThread->nodes.fetch_add(1, std::memory_order_relaxed);
        pos.do_move(move, st, givesCheck);

        // These reduction adjustments have proven non-linear scaling.
        // They are optimized to time controls of 180 + 1.8 and longer,
        // so changing them or adding conditions that are similar requires
        // tests at these types of time controls.

        // Decrease reduction if position is or has been on the PV (~7 Elo)
        if (ss->ttPv)
            r -= 1 + (ttData.value > alpha) + (ttData.depth >= depth);

        // Decrease reduction for PvNodes (~0 Elo on STC, ~2 Elo on LTC)
        if (PvNode)
            r--;

        // These reduction adjustments have no proven non-linear scaling

        // Increase reduction for cut nodes (~4 Elo)
        if (cutNode)
            r += 2 - (ttData.depth >= depth && ss->ttPv);

        // Increase reduction if ttMove is a capture (~3 Elo)
        if (ttCapture)
            r++;

        // Increase reduction if next ply has a lot of fail high (~5 Elo)
        if ((ss + 1)->cutoffCnt > 3)
            r += 1 + allNode;

        // For first picked move (ttMove) reduce reduction, but never allow
        // reduction to go below 0 (~3 Elo)
        else if (move == ttData.move)
            r = std::max(0, r - 2);

        ss->statScore = 2 * thisThread->mainHistory[us][move.from_to()]
                      + (*contHist[0])[movedPiece][move.to_sq()]
                      + (*contHist[1])[movedPiece][move.to_sq()] - 4664;

        // Decrease/increase reduction for moves with a good/bad history (~8 Elo)
        r -= ss->statScore / 10898;

        // Step 17. Late moves reduction / extension (LMR, ~117 Elo)
        if (depth >= 2 && moveCount > 1 + (rootNode && depth < 10))
        {
            // In general we want to cap the LMR depth search at newDepth, but when
            // reduction is negative, we allow this move a limited search extension
            // beyond the first move depth.
            // To prevent problems when the max value is less than the min value,
            // std::clamp has been replaced by a more robust implementation.
            Depth d = std::max(1, std::min(newDepth - r, newDepth + !allNode));

            value = -search<NonPV>(pos, ss + 1, -(alpha + 1), -alpha, d, true);

            // Do a full-depth search when reduced LMR search fails high
            if (value > alpha && d < newDepth)
            {
                // Adjust full-depth search based on LMR results - if the result was
                // good enough search deeper, if it was bad enough search shallower.
                const bool doDeeperSearch    = value > (bestValue + 35 + 2 * newDepth);  // (~1 Elo)
                const bool doShallowerSearch = value < bestValue + 8;                    // (~2 Elo)

                newDepth += doDeeperSearch - doShallowerSearch;

                if (newDepth > d)
                    value = -search<NonPV>(pos, ss + 1, -(alpha + 1), -alpha, newDepth, !cutNode);

                // Post LMR continuation history updates (~1 Elo)
                int bonus = value >= beta ? stat_bonus(newDepth) : -stat_malus(newDepth);

                update_continuation_histories(ss, movedPiece, move.to_sq(), bonus);
            }
        }

        // Step 18. Full-depth search when LMR is skipped
        else if (!PvNode || moveCount > 1)
        {
            // Increase reduction if ttMove is not present (~6 Elo)
            if (!ttData.move)
                r += 2;

            // Note that if expected reduction is high, we reduce search depth by 1 here (~9 Elo)
            value = -search<NonPV>(pos, ss + 1, -(alpha + 1), -alpha, newDepth - (r > 3), !cutNode);
        }

        // For PV nodes only, do a full PV search on the first move or after a fail high,
        // otherwise let the parent node fail low with value <= alpha and try another move.
        if (PvNode && (moveCount == 1 || value > alpha))
        {
            (ss + 1)->pv    = pv;
            (ss + 1)->pv[0] = Move::none();

            // Extend move from transposition table if we are about to dive into qsearch.
            if (move == ttData.move && ss->ply <= thisThread->rootDepth * 2)
                newDepth = std::max(newDepth, 1);

            value = -search<PV>(pos, ss + 1, -beta, -alpha, newDepth, false);
        }

        // Step 19. Undo move
        pos.undo_move(move);

        assert(value > -VALUE_INFINITE && value < VALUE_INFINITE);

        // Step 20. Check for a new best move
        // Finished searching the move. If a stop occurred, the return value of
        // the search cannot be trusted, and we return immediately without updating
        // best move, principal variation nor transposition table.
        if (threads.stop.load(std::memory_order_relaxed))
            return VALUE_ZERO;

        if (rootNode)
        {
            RootMove& rm =
              *std::find(thisThread->rootMoves.begin(), thisThread->rootMoves.end(), move);

            rm.effort += nodes - nodeCount;

            rm.averageScore =
              rm.averageScore != -VALUE_INFINITE ? (value + rm.averageScore) / 2 : value;

            // PV move or new best move?
            if (moveCount == 1 || value > alpha)
            {
                rm.score = rm.uciScore = value;
                rm.selDepth            = thisThread->selDepth;
                rm.scoreLowerbound = rm.scoreUpperbound = false;

                if (value >= beta)
                {
                    rm.scoreLowerbound = true;
                    rm.uciScore        = beta;
                }
                else if (value <= alpha)
                {
                    rm.scoreUpperbound = true;
                    rm.uciScore        = alpha;
                }

                rm.pv.resize(1);

                assert((ss + 1)->pv);

                for (Move* m = (ss + 1)->pv; *m != Move::none(); ++m)
                    rm.pv.push_back(*m);

                // We record how often the best move has been changed in each iteration.
                // This information is used for time management. In MultiPV mode,
                // we must take care to only do this for the first PV line.
                if (moveCount > 1 && !thisThread->pvIdx)
                    ++thisThread->bestMoveChanges;
            }
            else
                // All other moves but the PV, are set to the lowest value: this
                // is not a problem when sorting because the sort is stable and the
                // move position in the list is preserved - just the PV is pushed up.
                rm.score = -VALUE_INFINITE;
        }

        // In case we have an alternative move equal in eval to the current bestmove,
        // promote it to bestmove by pretending it just exceeds alpha (but not beta).
        int inc =
          (value == bestValue && (int(nodes) & 15) == 0 && ss->ply + 2 >= thisThread->rootDepth
           && std::abs(value) + 1 < VALUE_TB_WIN_IN_MAX_PLY);

        if (value + inc > bestValue)
        {
            bestValue = value;

            if (value + inc > alpha)
            {
                bestMove = move;

                if (PvNode && !rootNode)  // Update pv even in fail-high case
                    update_pv(ss->pv, move, (ss + 1)->pv);

                if (value >= beta)
                {
                    ss->cutoffCnt += !ttData.move + (extension < 2);
                    assert(value >= beta);  // Fail high
                    break;
                }
                else
                {
                    // Reduce other moves if we have found at least one score improvement (~2 Elo)
                    if (depth > 2 && depth < 14 && std::abs(value) < VALUE_TB_WIN_IN_MAX_PLY)
                        depth -= 2;

                    assert(depth > 0);
                    alpha = value;  // Update alpha! Always alpha < beta
                }
            }
        }

        // If the move is worse than some previously searched move,
        // remember it, to update its stats later.
        if (move != bestMove && moveCount <= 32)
        {
            if (capture)
                capturesSearched.push_back(move);
            else
                quietsSearched.push_back(move);
        }
    }

    // Step 21. Check for mate and stalemate
    // All legal moves have been searched and if there are no legal moves, it
    // must be a mate or a stalemate. If we are in a singular extension search then
    // return a fail low score.

    assert(moveCount || !ss->inCheck || excludedMove || !MoveList<LEGAL>(pos).size());

    // Adjust best value for fail high cases at non-pv nodes
    if (!PvNode && bestValue >= beta && std::abs(bestValue) < VALUE_TB_WIN_IN_MAX_PLY
        && std::abs(beta) < VALUE_TB_WIN_IN_MAX_PLY && std::abs(alpha) < VALUE_TB_WIN_IN_MAX_PLY)
        bestValue = (bestValue * depth + beta) / (depth + 1);

    if (!moveCount)
        bestValue = excludedMove ? alpha : ss->inCheck ? mated_in(ss->ply) : VALUE_DRAW;

    // If there is a move that produces search value greater than alpha,
    // we update the stats of searched moves.
    else if (bestMove)
        update_all_stats(pos, ss, *this, bestMove, prevSq, quietsSearched, capturesSearched, depth);

    // Bonus for prior countermove that caused the fail low
    else if (!priorCapture && prevSq != SQ_NONE)
    {
        int bonus = (122 * (depth > 5) + 39 * !allNode + 165 * ((ss - 1)->moveCount > 8)
                     + 107 * (!ss->inCheck && bestValue <= ss->staticEval - 98)
                     + 134 * (!(ss - 1)->inCheck && bestValue <= -(ss - 1)->staticEval - 91));

        // Proportional to "how much damage we have to undo"
        bonus += std::clamp(-(ss - 1)->statScore / 100, -94, 304);

        bonus = std::max(bonus, 0);

        update_continuation_histories(ss - 1, pos.piece_on(prevSq), prevSq,
                                      stat_bonus(depth) * bonus / 116);
        thisThread->mainHistory[~us][((ss - 1)->currentMove).from_to()]
          << stat_bonus(depth) * bonus / 180;

        if (type_of(pos.piece_on(prevSq)) != PAWN && ((ss - 1)->currentMove).type_of() != PROMOTION)
            thisThread->pawnHistory[pawn_structure_index(pos)][pos.piece_on(prevSq)][prevSq]
              << stat_bonus(depth) * bonus / 25;
    }

    // Bonus when search fails low and there is a TT move
    else if (moveCount > 1 && ttData.move && !allNode)
        thisThread->mainHistory[us][ttData.move.from_to()] << stat_bonus(depth) / 4;

    if (PvNode)
        bestValue = std::min(bestValue, maxValue);

    // If no good move is found and the previous position was ttPv, then the previous
    // opponent move is probably good and the new position is added to the search tree. (~7 Elo)
    if (bestValue <= alpha)
        ss->ttPv = ss->ttPv || ((ss - 1)->ttPv && depth > 3);

    // Write gathered information in transposition table. Note that the
    // static evaluation is saved as it was before correction history.
    if (!excludedMove && !(rootNode && thisThread->pvIdx))
        ttWriter.write(posKey, value_to_tt(bestValue, ss->ply), ss->ttPv,
                       bestValue >= beta    ? BOUND_LOWER
                       : PvNode && bestMove ? BOUND_EXACT
                                            : BOUND_UPPER,
                       depth, bestMove, unadjustedStaticEval, tt.generation());

    // Adjust correction history
    if (!ss->inCheck && (!bestMove || !pos.capture(bestMove))
        && !(bestValue >= beta && bestValue <= ss->staticEval)
        && !(!bestMove && bestValue >= ss->staticEval))
    {
        auto bonus = std::clamp(int(bestValue - ss->staticEval) * depth / 8,
                                -CORRECTION_HISTORY_LIMIT / 4, CORRECTION_HISTORY_LIMIT / 4);
        thisThread->correctionHistory[us][pawn_structure_index<Correction>(pos)] << bonus;
    }

    assert(bestValue > -VALUE_INFINITE && bestValue < VALUE_INFINITE);

    return bestValue;
}

// Quiescence search function, which is called by the main search function with
// depth zero, or recursively with further decreasing depth. With depth <= 0, we
// "should" be using static eval only, but tactical moves may confuse the static eval.
// To fight this horizon effect, we implement this qsearch of tactical moves (~155 Elo).
// See https://www.chessprogramming.org/Horizon_Effect
// and https://www.chessprogramming.org/Quiescence_Search
template<NodeType nodeType>
Value Search::Worker::qsearch(Position& pos, Stack* ss, Value alpha, Value beta) {

    static_assert(nodeType != Root);
    constexpr bool PvNode = nodeType == PV;

    assert(alpha >= -VALUE_INFINITE && alpha < beta && beta <= VALUE_INFINITE);
    assert(PvNode || (alpha == beta - 1));

    // Check if we have an upcoming move that draws by repetition (~1 Elo)
    if (alpha < VALUE_DRAW && pos.upcoming_repetition(ss->ply))
    {
        alpha = value_draw(this->nodes);
        if (alpha >= beta)
            return alpha;
    }

    Move      pv[MAX_PLY + 1];
    StateInfo st;
    ASSERT_ALIGNED(&st, Eval::NNUE::CacheLineSize);

    Key   posKey;
    Move  move, bestMove;
    Value bestValue, value, futilityBase;
    bool  pvHit, givesCheck, capture;
    int   moveCount;
    Color us = pos.side_to_move();

    // Step 1. Initialize node
    if (PvNode)
    {
        (ss + 1)->pv = pv;
        ss->pv[0]    = Move::none();
    }

    Worker* thisThread = this;
    bestMove           = Move::none();
    ss->inCheck        = pos.checkers();
    moveCount          = 0;

    // Used to send selDepth info to GUI (selDepth counts from 1, ply from 0)
    if (PvNode && thisThread->selDepth < ss->ply + 1)
        thisThread->selDepth = ss->ply + 1;

    // Step 2. Check for an immediate draw or maximum ply reached
    if (pos.is_draw(ss->ply) || ss->ply >= MAX_PLY)
        return (ss->ply >= MAX_PLY && !ss->inCheck)
               ? evaluate(networks[numaAccessToken], pos, refreshTable, thisThread->optimism[us])
               : VALUE_DRAW;

    assert(0 <= ss->ply && ss->ply < MAX_PLY);

    // Step 3. Transposition table lookup
    posKey                         = pos.key();
    auto [ttHit, ttData, ttWriter] = tt.probe(posKey);
    // Need further processing of the saved data
    ss->ttHit    = ttHit;
    ttData.move  = ttHit ? ttData.move : Move::none();
    ttData.value = ttHit ? value_from_tt(ttData.value, ss->ply, pos.rule50_count()) : VALUE_NONE;
    pvHit        = ttHit && ttData.is_pv;

    // At non-PV nodes we check for an early TT cutoff
    if (!PvNode && ttData.depth >= DEPTH_QS
        && ttData.value != VALUE_NONE  // Can happen when !ttHit or when access race in probe()
        && (ttData.bound & (ttData.value >= beta ? BOUND_LOWER : BOUND_UPPER)))
        return ttData.value;

    // Step 4. Static evaluation of the position
    Value unadjustedStaticEval = VALUE_NONE;
    if (ss->inCheck)
        bestValue = futilityBase = -VALUE_INFINITE;
    else
    {
        if (ss->ttHit)
        {
            // Never assume anything about values stored in TT
            unadjustedStaticEval = ttData.eval;
            if (unadjustedStaticEval == VALUE_NONE)
                unadjustedStaticEval =
                  evaluate(networks[numaAccessToken], pos, refreshTable, thisThread->optimism[us]);
            ss->staticEval = bestValue =
              to_corrected_static_eval(unadjustedStaticEval, *thisThread, pos);

            // ttValue can be used as a better position evaluation (~13 Elo)
            if (std::abs(ttData.value) < VALUE_TB_WIN_IN_MAX_PLY
                && (ttData.bound & (ttData.value > bestValue ? BOUND_LOWER : BOUND_UPPER)))
                bestValue = ttData.value;
        }
        else
        {
            // In case of null move search, use previous static eval with opposite sign
            unadjustedStaticEval =
              (ss - 1)->currentMove != Move::null()
                ? evaluate(networks[numaAccessToken], pos, refreshTable, thisThread->optimism[us])
                : -(ss - 1)->staticEval;
            ss->staticEval = bestValue =
              to_corrected_static_eval(unadjustedStaticEval, *thisThread, pos);
        }

        // Stand pat. Return immediately if static value is at least beta
        if (bestValue >= beta)
        {
            if (std::abs(bestValue) < VALUE_TB_WIN_IN_MAX_PLY)
                bestValue = (3 * bestValue + beta) / 4;
            if (!ss->ttHit)
                ttWriter.write(posKey, value_to_tt(bestValue, ss->ply), false, BOUND_LOWER,
                               DEPTH_UNSEARCHED, Move::none(), unadjustedStaticEval,
                               tt.generation());
            return bestValue;
        }

        if (bestValue > alpha)
            alpha = bestValue;

        futilityBase = ss->staticEval + 299;
    }

    const PieceToHistory* contHist[] = {(ss - 1)->continuationHistory,
                                        (ss - 2)->continuationHistory};

    Square prevSq = ((ss - 1)->currentMove).is_ok() ? ((ss - 1)->currentMove).to_sq() : SQ_NONE;

    // Initialize a MovePicker object for the current position, and prepare to search
    // the moves. We presently use two stages of move generator in quiescence search:
    // captures, or evasions only when in check.
    MovePicker mp(pos, ttData.move, DEPTH_QS, &thisThread->mainHistory, &thisThread->captureHistory,
                  contHist, &thisThread->pawnHistory);

    // Step 5. Loop through all pseudo-legal moves until no moves remain or a beta
    // cutoff occurs.
    while ((move = mp.next_move()) != Move::none())
    {
        assert(move.is_ok());

        if (!pos.legal(move))
            continue;

        givesCheck = pos.gives_check(move);
        capture    = pos.capture_stage(move);

        moveCount++;

        // Step 6. Pruning
        if (bestValue > VALUE_TB_LOSS_IN_MAX_PLY && pos.non_pawn_material(us))
        {
            // Futility pruning and moveCount pruning (~10 Elo)
            if (!givesCheck && move.to_sq() != prevSq && futilityBase > VALUE_TB_LOSS_IN_MAX_PLY
                && move.type_of() != PROMOTION)
            {
                if (moveCount > 2)
                    continue;

                Value futilityValue = futilityBase + PieceValue[pos.piece_on(move.to_sq())];

                // If static eval + value of piece we are going to capture is
                // much lower than alpha, we can prune this move. (~2 Elo)
                if (futilityValue <= alpha)
                {
                    bestValue = std::max(bestValue, futilityValue);
                    continue;
                }

                // If static eval is much lower than alpha and move is
                // not winning material, we can prune this move. (~2 Elo)
                if (futilityBase <= alpha && !pos.see_ge(move, 1))
                {
                    bestValue = std::max(bestValue, futilityBase);
                    continue;
                }

                // If static exchange evaluation is much worse than what
                // is needed to not fall below alpha, we can prune this move.
                if (futilityBase > alpha && !pos.see_ge(move, (alpha - futilityBase) * 4))
                {
                    bestValue = alpha;
                    continue;
                }
            }
            // Continuation history based pruning (~2 Elo)
            if (!capture
                && (*contHist[0])[pos.moved_piece(move)][move.to_sq()]
                       + (*contHist[1])[pos.moved_piece(move)][move.to_sq()]
                       + thisThread->pawnHistory[pawn_structure_index(pos)][pos.moved_piece(move)]
                                                [move.to_sq()]
                     <= 4643)
                continue;

            // Do not search moves with bad enough SEE values (~5 Elo)
            if (!pos.see_ge(move, -83))
                continue;
        }

        // Speculative prefetch as early as possible
        prefetch(tt.first_entry(pos.key_after(move)));

        // Update the current move
        ss->currentMove = move;
        ss->continuationHistory =
          &thisThread
             ->continuationHistory[ss->inCheck][capture][pos.moved_piece(move)][move.to_sq()];

        // Step 7. Make and search the move
        thisThread->nodes.fetch_add(1, std::memory_order_relaxed);
        pos.do_move(move, st, givesCheck);
        value = -qsearch<nodeType>(pos, ss + 1, -beta, -alpha);
        pos.undo_move(move);

        assert(value > -VALUE_INFINITE && value < VALUE_INFINITE);

        // Step 8. Check for a new best move
        if (value > bestValue)
        {
            bestValue = value;

            if (value > alpha)
            {
                bestMove = move;

                if (PvNode)  // Update pv even in fail-high case
                    update_pv(ss->pv, move, (ss + 1)->pv);

                if (value < beta)  // Update alpha here!
                    alpha = value;
                else
                    break;  // Fail high
            }
        }
    }

    // Step 9. Check for mate
    // All legal moves have been searched. A special case: if we are
    // in check and no legal moves were found, it is checkmate.
    if (ss->inCheck && bestValue == -VALUE_INFINITE)
    {
        assert(!MoveList<LEGAL>(pos).size());
        return mated_in(ss->ply);  // Plies to mate from the root
    }

    if (std::abs(bestValue) < VALUE_TB_WIN_IN_MAX_PLY && bestValue >= beta)
        bestValue = (3 * bestValue + beta) / 4;

    // Save gathered info in transposition table. The static evaluation
    // is saved as it was before adjustment by correction history.
    ttWriter.write(posKey, value_to_tt(bestValue, ss->ply), pvHit,
                   bestValue >= beta ? BOUND_LOWER : BOUND_UPPER, DEPTH_QS, bestMove,
                   unadjustedStaticEval, tt.generation());

    assert(bestValue > -VALUE_INFINITE && bestValue < VALUE_INFINITE);

    return bestValue;
}

Depth Search::Worker::reduction(bool i, Depth d, int mn, int delta) const {
    int reductionScale = reductions[d] * reductions[mn];
    return (reductionScale + 1274 - delta * 746 / rootDelta) / 1024 + (!i && reductionScale > 1293);
}

// elapsed() returns the time elapsed since the search started. If the
// 'nodestime' option is enabled, it will return the count of nodes searched
// instead. This function is called to check whether the search should be
// stopped based on predefined thresholds like time limits or nodes searched.
//
// elapsed_time() returns the actual time elapsed since the start of the search.
// This function is intended for use only when printing PV outputs, and not used
// for making decisions within the search algorithm itself.
TimePoint Search::Worker::elapsed() const {
    return main_manager()->tm.elapsed([this]() { return threads.nodes_searched(); });
}

TimePoint Search::Worker::elapsed_time() const { return main_manager()->tm.elapsed_time(); }

namespace {
// Adjusts a mate or TB score from "plies to mate from the root" to
// "plies to mate from the current position". Standard scores are unchanged.
// The function is called before storing a value in the transposition table.
Value value_to_tt(Value v, int ply) {

    assert(v != VALUE_NONE);
    return v >= VALUE_TB_WIN_IN_MAX_PLY ? v + ply : v <= VALUE_TB_LOSS_IN_MAX_PLY ? v - ply : v;
}

// Inverse of value_to_tt(): it adjusts a mate or TB score from the transposition
// table (which refers to the plies to mate/be mated from current position) to
// "plies to mate/be mated (TB win/loss) from the root". However, to avoid
// potentially false mate or TB scores related to the 50 moves rule and the
// graph history interaction, we return the highest non-TB score instead.
Value value_from_tt(Value v, int ply, int r50c) {

    if (v == VALUE_NONE)
        return VALUE_NONE;

    // handle TB win or better
    if (v >= VALUE_TB_WIN_IN_MAX_PLY)
    {
        // Downgrade a potentially false mate score
        if (v >= VALUE_MATE_IN_MAX_PLY && VALUE_MATE - v > 100 - r50c)
            return VALUE_TB_WIN_IN_MAX_PLY - 1;

        // Downgrade a potentially false TB score.
        if (VALUE_TB - v > 100 - r50c)
            return VALUE_TB_WIN_IN_MAX_PLY - 1;

        return v - ply;
    }

    // handle TB loss or worse
    if (v <= VALUE_TB_LOSS_IN_MAX_PLY)
    {
        // Downgrade a potentially false mate score.
        if (v <= VALUE_MATED_IN_MAX_PLY && VALUE_MATE + v > 100 - r50c)
            return VALUE_TB_LOSS_IN_MAX_PLY + 1;

        // Downgrade a potentially false TB score.
        if (VALUE_TB + v > 100 - r50c)
            return VALUE_TB_LOSS_IN_MAX_PLY + 1;

        return v + ply;
    }

    return v;
}

// Adds current move and appends child pv[]
void update_pv(Move* pv, Move move, const Move* childPv) {

    for (*pv++ = move; childPv && *childPv != Move::none();)
        *pv++ = *childPv++;
    *pv = Move::none();
}

// Updates stats at the end of search() when a bestMove is found
void update_all_stats(const Position&      pos,
                      Stack*               ss,
                      Search::Worker&      workerThread,
                      Move                 bestMove,
                      Square               prevSq,
                      ValueList<Move, 32>& quietsSearched,
                      ValueList<Move, 32>& capturesSearched,
                      Depth                depth) {

    CapturePieceToHistory& captureHistory = workerThread.captureHistory;
    Piece                  moved_piece    = pos.moved_piece(bestMove);
    PieceType              captured;

    int quietMoveBonus = stat_bonus(depth);
    int quietMoveMalus = stat_malus(depth);

    if (!pos.capture_stage(bestMove))
    {
        update_quiet_histories(pos, ss, workerThread, bestMove, quietMoveBonus);

        // Decrease stats for all non-best quiet moves
        for (Move move : quietsSearched)
            update_quiet_histories(pos, ss, workerThread, move, -quietMoveMalus);
    }
    else
    {
        // Increase stats for the best move in case it was a capture move
        captured = type_of(pos.piece_on(bestMove.to_sq()));
        captureHistory[moved_piece][bestMove.to_sq()][captured] << quietMoveBonus;
    }

    // Extra penalty for a quiet early move that was not a TT move in
    // previous ply when it gets refuted.
    if (prevSq != SQ_NONE && ((ss - 1)->moveCount == 1 + (ss - 1)->ttHit) && !pos.captured_piece())
        update_continuation_histories(ss - 1, pos.piece_on(prevSq), prevSq, -quietMoveMalus);

    // Decrease stats for all non-best capture moves
    for (Move move : capturesSearched)
    {
        moved_piece = pos.moved_piece(move);
        captured    = type_of(pos.piece_on(move.to_sq()));
        captureHistory[moved_piece][move.to_sq()][captured] << -quietMoveMalus;
    }
}

// Updates histories of the move pairs formed by moves
// at ply -1, -2, -3, -4, and -6 with current move.
void update_continuation_histories(Stack* ss, Piece pc, Square to, int bonus) {

    bonus = bonus * 52 / 64;

    for (int i : {1, 2, 3, 4, 6})
    {
        // Only update the first 2 continuation histories if we are in check
        if (ss->inCheck && i > 2)
            break;
        if (((ss - i)->currentMove).is_ok())
            (*(ss - i)->continuationHistory)[pc][to] << bonus / (1 + (i == 3));
    }
}

// Updates move sorting heuristics

void update_quiet_histories(
  const Position& pos, Stack* ss, Search::Worker& workerThread, Move move, int bonus) {

    Color us = pos.side_to_move();
    workerThread.mainHistory[us][move.from_to()] << bonus;

    update_continuation_histories(ss, pos.moved_piece(move), move.to_sq(), bonus);

    int pIndex = pawn_structure_index(pos);
    workerThread.pawnHistory[pIndex][pos.moved_piece(move)][move.to_sq()] << bonus / 2;
}

}

// When playing with strength handicap, choose the best move among a set of
// RootMoves using a statistical rule dependent on 'level'. Idea by Heinz van Saanen.
Move Skill::pick_best(const RootMoves& rootMoves, size_t multiPV) {
    static PRNG rng(now());  // PRNG sequence should be non-deterministic

    // RootMoves are already sorted by score in descending order
    Value  topScore = rootMoves[0].score;
    int    delta    = std::min(topScore - rootMoves[multiPV - 1].score, int(PawnValue));
    int    maxScore = -VALUE_INFINITE;
    double weakness = 120 - 2 * level;

    // Choose best move. For each move score we add two terms, both dependent on
    // weakness. One is deterministic and bigger for weaker levels, and one is
    // random. Then we choose the move with the resulting highest score.
    for (size_t i = 0; i < multiPV; ++i)
    {
        // This is our magic formula
        int push = (weakness * int(topScore - rootMoves[i].score)
                    + delta * (rng.rand<unsigned>() % int(weakness)))
                 / 128;

        if (rootMoves[i].score + push >= maxScore)
        {
            maxScore = rootMoves[i].score + push;
            best     = rootMoves[i].pv[0];
        }
    }

    return best;
}

// Used to print debug info and, more importantly, to detect
// when we are out of available time and thus stop the search.
void SearchManager::check_time(Search::Worker& worker) {
    if (--callsCnt > 0)
        return;

    // When using nodes, ensure checking rate is not lower than 0.1% of nodes
    callsCnt = worker.limits.nodes ? std::min(512, int(worker.limits.nodes / 1024)) : 512;

    static TimePoint lastInfoTime = now();

    TimePoint elapsed = tm.elapsed([&worker]() { return worker.threads.nodes_searched(); });
    TimePoint tick    = worker.limits.startTime + elapsed;

    if (tick - lastInfoTime >= 1000)
    {
        lastInfoTime = tick;
        dbg_print();
    }

    // We should not stop pondering until told so by the GUI
    if (ponder)
        return;

    if (
      // Later we rely on the fact that we can at least use the mainthread previous
      // root-search score and PV in a multithreaded environment to prove mated-in scores.
      worker.completedDepth >= 1
      && ((worker.limits.use_time_management() && (elapsed > tm.maximum() || stopOnPonderhit))
          || (worker.limits.movetime && elapsed >= worker.limits.movetime)
          || (worker.limits.nodes && worker.threads.nodes_searched() >= worker.limits.nodes)))
        worker.threads.stop = worker.threads.abortedSearch = true;
}

// Used to correct and extend PVs for moves that have a TB (but not a mate) score.
// Keeps the search based PV for as long as it is verified to maintain the game
// outcome, truncates afterwards. Finally, extends to mate the PV, providing a
// possible continuation (but not a proven mating line).
void syzygy_extend_pv(const OptionsMap&         options,
                      const Search::LimitsType& limits,
                      Position&                 pos,
                      RootMove&                 rootMove,
                      Value&                    v) {

    auto t_start      = std::chrono::steady_clock::now();
    int  moveOverhead = int(options["Move Overhead"]);

    // Do not use more than moveOverhead / 2 time, if time management is active
    auto time_abort = [&t_start, &moveOverhead, &limits]() -> bool {
        auto t_end = std::chrono::steady_clock::now();
        return limits.use_time_management()
            && 2 * std::chrono::duration<double, std::milli>(t_end - t_start).count()
                 > moveOverhead;
    };

    std::list<StateInfo> sts;

    // Step 0, do the rootMove, no correction allowed, as needed for MultiPV in TB.
    auto& stRoot = sts.emplace_back();
    pos.do_move(rootMove.pv[0], stRoot);
    int ply = 1;

    // Step 1, walk the PV to the last position in TB with correct decisive score
    while (size_t(ply) < rootMove.pv.size())
    {
        Move& pvMove = rootMove.pv[ply];

        RootMoves legalMoves;
        for (const auto& m : MoveList<LEGAL>(pos))
            legalMoves.emplace_back(m);

        Tablebases::Config config = Tablebases::rank_root_moves(options, pos, legalMoves);
        RootMove&          rm     = *std::find(legalMoves.begin(), legalMoves.end(), pvMove);

        if (legalMoves[0].tbRank != rm.tbRank)
            break;

        ply++;

        auto& st = sts.emplace_back();
        pos.do_move(pvMove, st);

        // Do not allow for repetitions or drawing moves along the PV in TB regime
        if (config.rootInTB && pos.is_draw(ply))
        {
            pos.undo_move(pvMove);
            ply--;
            break;
        }

        // Full PV shown will thus be validated and end in TB.
        // If we cannot validate the full PV in time, we do not show it.
        if (config.rootInTB && time_abort())
            break;
    }

    // Resize the PV to the correct part
    rootMove.pv.resize(ply);

    // Step 2, now extend the PV to mate, as if the user explored syzygy-tables.info
    // using top ranked moves (minimal DTZ), which gives optimal mates only for simple
    // endgames e.g. KRvK.
    while (!(pos.is_draw(0)))
    {
        if (time_abort())
            break;

        RootMoves legalMoves;
        for (const auto& m : MoveList<LEGAL>(pos))
        {
            auto&     rm = legalMoves.emplace_back(m);
            StateInfo tmpSI;
            pos.do_move(m, tmpSI);
            // Give a score of each move to break DTZ ties restricting opponent mobility,
            // but not giving the opponent a capture.
            for (const auto& mOpp : MoveList<LEGAL>(pos))
                rm.tbRank -= pos.capture(mOpp) ? 100 : 1;
            pos.undo_move(m);
        }

        // Mate found
        if (legalMoves.size() == 0)
            break;

        // Sort moves according to their above assigned rank.
        // This will break ties for moves with equal DTZ in rank_root_moves.
        std::stable_sort(
          legalMoves.begin(), legalMoves.end(),
          [](const Search::RootMove& a, const Search::RootMove& b) { return a.tbRank > b.tbRank; });

        // The winning side tries to minimize DTZ, the losing side maximizes it
        Tablebases::Config config = Tablebases::rank_root_moves(options, pos, legalMoves, true);

        // If DTZ is not available we might not find a mate, so we bail out
        if (!config.rootInTB || config.cardinality > 0)
            break;

        ply++;

        Move& pvMove = legalMoves[0].pv[0];
        rootMove.pv.push_back(pvMove);
        auto& st = sts.emplace_back();
        pos.do_move(pvMove, st);
    }

    // Finding a draw in this function is an exceptional case, that cannot happen
    // during engine game play, since we have a winning score, and play correctly
    // with TB support. However, it can be that a position is draw due to the 50 move
    // rule if it has been been reached on the board with a non-optimal 50 move counter
    // (e.g. 8/8/6k1/3B4/3K4/4N3/8/8 w - - 54 106 ) which TB with dtz counter rounding
    // cannot always correctly rank. See also
    // https://github.com/official-stockfish/Stockfish/issues/5175#issuecomment-2058893495
    // We adjust the score to match the found PV. Note that a TB loss score can be
    // displayed if the engine did not find a drawing move yet, but eventually search
    // will figure it out (e.g. 1kq5/q2r4/5K2/8/8/8/8/7Q w - - 96 1 )
    if (pos.is_draw(0))
        v = VALUE_DRAW;

    // Undo the PV moves
    for (auto it = rootMove.pv.rbegin(); it != rootMove.pv.rend(); ++it)
        pos.undo_move(*it);

    // Inform if we couldn't get a full extension in time
    if (time_abort())
        sync_cout
          << "info string Syzygy based PV extension requires more time, increase Move Overhead as needed."
          << sync_endl;
}

void SearchManager::pv(Search::Worker&           worker,
                       const ThreadPool&         threads,
                       const TranspositionTable& tt,
                       Depth                     depth) {

    const auto nodes     = threads.nodes_searched();
    auto&      rootMoves = worker.rootMoves;
    auto&      pos       = worker.rootPos;
    size_t     pvIdx     = worker.pvIdx;
    size_t     multiPV   = std::min(size_t(worker.options["MultiPV"]), rootMoves.size());
    uint64_t   tbHits    = threads.tb_hits() + (worker.tbConfig.rootInTB ? rootMoves.size() : 0);

    for (size_t i = 0; i < multiPV; ++i)
    {
        bool updated = rootMoves[i].score != -VALUE_INFINITE;

        if (depth == 1 && !updated && i > 0)
            continue;

        Depth d = updated ? depth : std::max(1, depth - 1);
        Value v = updated ? rootMoves[i].uciScore : rootMoves[i].previousScore;

        if (v == -VALUE_INFINITE)
            v = VALUE_ZERO;

        bool tb = worker.tbConfig.rootInTB && std::abs(v) <= VALUE_TB;
        v       = tb ? rootMoves[i].tbScore : v;

        bool isExact = i != pvIdx || tb || !updated;  // tablebase- and previous-scores are exact

        // Potentially correct and extend the PV, and in exceptional cases v
        if (std::abs(v) >= VALUE_TB_WIN_IN_MAX_PLY && std::abs(v) < VALUE_MATE_IN_MAX_PLY
            && ((!rootMoves[i].scoreLowerbound && !rootMoves[i].scoreUpperbound) || isExact))
            syzygy_extend_pv(worker.options, worker.limits, pos, rootMoves[i], v);

        std::string pv;
        for (Move m : rootMoves[i].pv)
            pv += UCIEngine::move(m, pos.is_chess960()) + " ";

        // Remove last whitespace
        if (!pv.empty())
            pv.pop_back();

        auto wdl   = worker.options["UCI_ShowWDL"] ? UCIEngine::wdl(v, pos) : "";
        auto bound = rootMoves[i].scoreLowerbound
                     ? "lowerbound"
                     : (rootMoves[i].scoreUpperbound ? "upperbound" : "");

        InfoFull info;

        info.depth    = d;
        info.selDepth = rootMoves[i].selDepth;
        info.multiPV  = i + 1;
        info.score    = {v, pos};
        info.wdl      = wdl;

        if (!isExact)
            info.bound = bound;

        TimePoint time = tm.elapsed_time() + 1;
        info.timeMs    = time;
        info.nodes     = nodes;
        info.nps       = nodes * 1000 / time;
        info.tbHits    = tbHits;
        info.pv        = pv;
        info.hashfull  = tt.hashfull();

        updates.onUpdateFull(info);
    }
}

// Called in case we have no ponder move before exiting the search,
// for instance, in case we stop the search during a fail high at root.
// We try hard to have a ponder move to return to the GUI,
// otherwise in case of 'ponder on' we have nothing to think about.
bool RootMove::extract_ponder_from_tt(const TranspositionTable& tt, Position& pos) {

    StateInfo st;
    ASSERT_ALIGNED(&st, Eval::NNUE::CacheLineSize);

    assert(pv.size() == 1);
    if (pv[0] == Move::none())
        return false;

    pos.do_move(pv[0], st);

    auto [ttHit, ttData, ttWriter] = tt.probe(pos.key());
    if (ttHit)
    {
        if (MoveList<LEGAL>(pos).contains(ttData.move))
            pv.push_back(ttData.move);
    }

    pos.undo_move(pv[0]);
    return pv.size() > 1;
}

}  // namespace Stockfish
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

// Constructor launches the thread and waits until it goes to sleep
// in idle_loop(). Note that 'searching' and 'exit' should be already set.
Thread::Thread(Search::SharedState&                    sharedState,
               std::unique_ptr<Search::ISearchManager> sm,
               size_t                                  n,
               OptionalThreadToNumaNodeBinder          binder) :
    idx(n),
    nthreads(sharedState.options["Threads"]),
    stdThread(&Thread::idle_loop, this) {

    wait_for_search_finished();

    run_custom_job([this, &binder, &sharedState, &sm, n]() {
        // Use the binder to [maybe] bind the threads to a NUMA node before doing
        // the Worker allocation. Ideally we would also allocate the SearchManager
        // here, but that's minor.
        this->numaAccessToken = binder();
        this->worker =
          std::make_unique<Search::Worker>(sharedState, std::move(sm), n, this->numaAccessToken);
    });

    wait_for_search_finished();
}

// Destructor wakes up the thread in idle_loop() and waits
// for its termination. Thread should be already waiting.
Thread::~Thread() {

    assert(!searching);

    exit = true;
    start_searching();
    stdThread.join();
}

// Wakes up the thread that will start the search
void Thread::start_searching() {
    assert(worker != nullptr);
    run_custom_job([this]() { worker->start_searching(); });
}

// Clears the histories for the thread worker (usually before a new game)
void Thread::clear_worker() {
    assert(worker != nullptr);
    run_custom_job([this]() { worker->clear(); });
}

// Blocks on the condition variable until the thread has finished searching
void Thread::wait_for_search_finished() {

    std::unique_lock<std::mutex> lk(mutex);
    cv.wait(lk, [&] { return !searching; });
}

// Launching a function in the thread
void Thread::run_custom_job(std::function<void()> f) {
    {
        std::unique_lock<std::mutex> lk(mutex);
        cv.wait(lk, [&] { return !searching; });
        jobFunc   = std::move(f);
        searching = true;
    }
    cv.notify_one();
}

void Thread::ensure_network_replicated() { worker->ensure_network_replicated(); }

// Thread gets parked here, blocked on the condition variable
// when the thread has no work to do.

void Thread::idle_loop() {
    while (true)
    {
        std::unique_lock<std::mutex> lk(mutex);
        searching = false;
        cv.notify_one();  // Wake up anyone waiting for search finished
        cv.wait(lk, [&] { return searching; });

        if (exit)
            return;

        std::function<void()> job = std::move(jobFunc);
        jobFunc                   = nullptr;

        lk.unlock();

        if (job)
            job();
    }
}

Search::SearchManager* ThreadPool::main_manager() { return main_thread()->worker->main_manager(); }

uint64_t ThreadPool::nodes_searched() const { return accumulate(&Search::Worker::nodes); }
uint64_t ThreadPool::tb_hits() const { return accumulate(&Search::Worker::tbHits); }

// Creates/destroys threads to match the requested number.
// Created and launched threads will immediately go to sleep in idle_loop.
// Upon resizing, threads are recreated to allow for binding if necessary.
void ThreadPool::set(const NumaConfig&                           numaConfig,
                     Search::SharedState                         sharedState,
                     const Search::SearchManager::UpdateContext& updateContext) {

    if (threads.size() > 0)  // destroy any existing thread(s)
    {
        main_thread()->wait_for_search_finished();

        threads.clear();

        boundThreadToNumaNode.clear();
    }

    const size_t requested = sharedState.options["Threads"];

    if (requested > 0)  // create new thread(s)
    {
        // Binding threads may be problematic when there's multiple NUMA nodes and
        // multiple Stockfish instances running. In particular, if each instance
        // runs a single thread then they would all be mapped to the first NUMA node.
        // This is undesirable, and so the default behaviour (i.e. when the user does not
        // change the NumaConfig UCI setting) is to not bind the threads to processors
        // unless we know for sure that we span NUMA nodes and replication is required.
        const std::string numaPolicy(sharedState.options["NumaPolicy"]);
        const bool        doBindThreads = [&]() {
            if (numaPolicy == "none")
                return false;

            if (numaPolicy == "auto")
                return numaConfig.suggests_binding_threads(requested);

            // numaPolicy == "system", or explicitly set by the user
            return true;
        }();

        boundThreadToNumaNode = doBindThreads
                                ? numaConfig.distribute_threads_among_numa_nodes(requested)
                                : std::vector<NumaIndex>{};

        while (threads.size() < requested)
        {
            const size_t    threadId = threads.size();
            const NumaIndex numaId   = doBindThreads ? boundThreadToNumaNode[threadId] : 0;
            auto            manager  = threadId == 0 ? std::unique_ptr<Search::ISearchManager>(
                                                        std::make_unique<Search::SearchManager>(updateContext))
                                                     : std::make_unique<Search::NullSearchManager>();

            // When not binding threads we want to force all access to happen
            // from the same NUMA node, because in case of NUMA replicated memory
            // accesses we don't want to trash cache in case the threads get scheduled
            // on the same NUMA node.
            auto binder = doBindThreads ? OptionalThreadToNumaNodeBinder(numaConfig, numaId)
                                        : OptionalThreadToNumaNodeBinder(numaId);

            threads.emplace_back(
              std::make_unique<Thread>(sharedState, std::move(manager), threadId, binder));
        }

        clear();

        main_thread()->wait_for_search_finished();
    }
}

// Sets threadPool data to initial values
void ThreadPool::clear() {
    if (threads.size() == 0)
        return;

    for (auto&& th : threads)
        th->clear_worker();

    for (auto&& th : threads)
        th->wait_for_search_finished();

    // These two affect the time taken on the first move of a game:
    main_manager()->bestPreviousAverageScore = VALUE_INFINITE;
    main_manager()->previousTimeReduction    = 0.85;

    main_manager()->callsCnt           = 0;
    main_manager()->bestPreviousScore  = VALUE_INFINITE;
    main_manager()->originalTimeAdjust = -1;
    main_manager()->tm.clear();
}

void ThreadPool::run_on_thread(size_t threadId, std::function<void()> f) {
    assert(threads.size() > threadId);
    threads[threadId]->run_custom_job(std::move(f));
}

void ThreadPool::wait_on_thread(size_t threadId) {
    assert(threads.size() > threadId);
    threads[threadId]->wait_for_search_finished();
}

size_t ThreadPool::num_threads() const { return threads.size(); }

// Wakes up main thread waiting in idle_loop() and returns immediately.
// Main thread will wake up other threads and start the search.
void ThreadPool::start_thinking(const OptionsMap&  options,
                                Position&          pos,
                                StateListPtr&      states,
                                Search::LimitsType limits) {

    main_thread()->wait_for_search_finished();

    main_manager()->stopOnPonderhit = stop = abortedSearch = false;
    main_manager()->ponder                                 = limits.ponderMode;

    increaseDepth = true;

    Search::RootMoves rootMoves;
    const auto        legalmoves = MoveList<LEGAL>(pos);

    for (const auto& uciMove : limits.searchmoves)
    {
        auto move = UCIEngine::to_move(pos, uciMove);

        if (std::find(legalmoves.begin(), legalmoves.end(), move) != legalmoves.end())
            rootMoves.emplace_back(move);
    }

    if (rootMoves.empty())
        for (const auto& m : legalmoves)
            rootMoves.emplace_back(m);

    Tablebases::Config tbConfig = Tablebases::rank_root_moves(options, pos, rootMoves);

    // After ownership transfer 'states' becomes empty, so if we stop the search
    // and call 'go' again without setting a new position states.get() == nullptr.
    assert(states.get() || setupStates.get());

    if (states.get())
        setupStates = std::move(states);  // Ownership transfer, states is now empty

    // We use Position::set() to set root position across threads. But there are
    // some StateInfo fields (previous, pliesFromNull, capturedPiece) that cannot
    // be deduced from a fen string, so set() clears them and they are set from
    // setupStates->back() later. The rootState is per thread, earlier states are
    // shared since they are read-only.
    for (auto&& th : threads)
    {
        th->run_custom_job([&]() {
            th->worker->limits = limits;
            th->worker->nodes = th->worker->tbHits = th->worker->nmpMinPly =
              th->worker->bestMoveChanges          = 0;
            th->worker->rootDepth = th->worker->completedDepth = 0;
            th->worker->rootMoves                              = rootMoves;
            th->worker->rootPos.set(pos.fen(), pos.is_chess960(), &th->worker->rootState);
            th->worker->rootState = setupStates->back();
            th->worker->tbConfig  = tbConfig;
        });
    }

    for (auto&& th : threads)
        th->wait_for_search_finished();

    main_thread()->start_searching();
}

Thread* ThreadPool::get_best_thread() const {

    Thread* bestThread = threads.front().get();
    Value   minScore   = VALUE_NONE;

    std::unordered_map<Move, int64_t, Move::MoveHash> votes(
      2 * std::min(size(), bestThread->worker->rootMoves.size()));

    // Find the minimum score of all threads
    for (auto&& th : threads)
        minScore = std::min(minScore, th->worker->rootMoves[0].score);

    // Vote according to score and depth, and select the best thread
    auto thread_voting_value = [minScore](Thread* th) {
        return (th->worker->rootMoves[0].score - minScore + 14) * int(th->worker->completedDepth);
    };

    for (auto&& th : threads)
        votes[th->worker->rootMoves[0].pv[0]] += thread_voting_value(th.get());

    for (auto&& th : threads)
    {
        const auto bestThreadScore = bestThread->worker->rootMoves[0].score;
        const auto newThreadScore  = th->worker->rootMoves[0].score;

        const auto& bestThreadPV = bestThread->worker->rootMoves[0].pv;
        const auto& newThreadPV  = th->worker->rootMoves[0].pv;

        const auto bestThreadMoveVote = votes[bestThreadPV[0]];
        const auto newThreadMoveVote  = votes[newThreadPV[0]];

        const bool bestThreadInProvenWin = bestThreadScore >= VALUE_TB_WIN_IN_MAX_PLY;
        const bool newThreadInProvenWin  = newThreadScore >= VALUE_TB_WIN_IN_MAX_PLY;

        const bool bestThreadInProvenLoss =
          bestThreadScore != -VALUE_INFINITE && bestThreadScore <= VALUE_TB_LOSS_IN_MAX_PLY;
        const bool newThreadInProvenLoss =
          newThreadScore != -VALUE_INFINITE && newThreadScore <= VALUE_TB_LOSS_IN_MAX_PLY;

        // We make sure not to pick a thread with truncated principal variation
        const bool betterVotingValue =
          thread_voting_value(th.get()) * int(newThreadPV.size() > 2)
          > thread_voting_value(bestThread) * int(bestThreadPV.size() > 2);

        if (bestThreadInProvenWin)
        {
            // Make sure we pick the shortest mate / TB conversion
            if (newThreadScore > bestThreadScore)
                bestThread = th.get();
        }
        else if (bestThreadInProvenLoss)
        {
            // Make sure we pick the shortest mated / TB conversion
            if (newThreadInProvenLoss && newThreadScore < bestThreadScore)
                bestThread = th.get();
        }
        else if (newThreadInProvenWin || newThreadInProvenLoss
                 || (newThreadScore > VALUE_TB_LOSS_IN_MAX_PLY
                     && (newThreadMoveVote > bestThreadMoveVote
                         || (newThreadMoveVote == bestThreadMoveVote && betterVotingValue))))
            bestThread = th.get();
    }

    return bestThread;
}

// Start non-main threads.
// Will be invoked by main thread after it has started searching.
void ThreadPool::start_searching() {

    for (auto&& th : threads)
        if (th != threads.front())
            th->start_searching();
}

// Wait for non-main threads
void ThreadPool::wait_for_search_finished() const {

    for (auto&& th : threads)
        if (th != threads.front())
            th->wait_for_search_finished();
}

std::vector<size_t> ThreadPool::get_bound_thread_count_by_numa_node() const {
    std::vector<size_t> counts;

    if (!boundThreadToNumaNode.empty())
    {
        NumaIndex highestNumaNode = 0;
        for (NumaIndex n : boundThreadToNumaNode)
            if (n > highestNumaNode)
                highestNumaNode = n;

        counts.resize(highestNumaNode + 1, 0);

        for (NumaIndex n : boundThreadToNumaNode)
            counts[n] += 1;
    }

    return counts;
}

void ThreadPool::ensure_network_replicated() {
    for (auto&& th : threads)
        th->ensure_network_replicated();
}

}  // namespace Stockfish
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

TimePoint TimeManagement::optimum() const { return optimumTime; }
TimePoint TimeManagement::maximum() const { return maximumTime; }

void TimeManagement::clear() {
    availableNodes = -1;  // When in 'nodes as time' mode
```

```
}

void TimeManagement::advance_nodes_time(std::int64_t nodes) {
    assert(useNodesTime);
    availableNodes = std::max(int64_t(0), availableNodes - nodes);
}

// Called at the beginning of the search and calculates
// the bounds of time allowed for the current game ply. We currently support:
//      1) x basetime (+ z increment)
//      2) x moves in y seconds (+ z increment)
void TimeManagement::init(Search::LimitsType& limits,
                          Color               us,
                          int                 ply,
                          const OptionsMap&   options,
                          double&             originalTimeAdjust) {
    TimePoint npmsec = TimePoint(options["nodestime"]);

    // If we have no time, we don't need to fully initialize TM.
    // startTime is used by movetime and useNodesTime is used in elapsed calls.
    startTime    = limits.startTime;
    useNodesTime = npmsec != 0;

    if (limits.time[us] == 0)
        return;

    TimePoint moveOverhead = TimePoint(options["Move Overhead"]);

    // optScale is a percentage of available time to use for the current move.
    // maxScale is a multiplier applied to optimumTime.
    double optScale, maxScale;

    // If we have to play in 'nodes as time' mode, then convert from time
    // to nodes, and use resulting values in time management formulas.
    // WARNING: to avoid time losses, the given npmsec (nodes per millisecond)
    // must be much lower than the real engine speed.
    if (useNodesTime)
    {
        if (availableNodes == -1)                       // Only once at game start
            availableNodes = npmsec * limits.time[us];  // Time is in msec

        // Convert from milliseconds to nodes
        limits.time[us] = TimePoint(availableNodes);
        limits.inc[us] *= npmsec;
        limits.npmsec = npmsec;
        moveOverhead *= npmsec;
    }

    // These numbers are used where multiplications, divisions or comparisons
    // with constants are involved.
    const int64_t   scaleFactor = useNodesTime ? npmsec : 1;
    const TimePoint scaledTime  = limits.time[us] / scaleFactor;
    const TimePoint scaledInc   = limits.inc[us] / scaleFactor;

    // Maximum move horizon of 50 moves
    int mtg = limits.movestogo ? std::min(limits.movestogo, 50) : 50;

    // If less than one second, gradually reduce mtg
    if (scaledTime < 1000 && double(mtg) / scaledInc > 0.05)
    {
        mtg = scaledTime * 0.05;
    }

    // Make sure timeLeft is > 0 since we may use it as a divisor
    TimePoint timeLeft = std::max(TimePoint(1), limits.time[us] + limits.inc[us] * (mtg - 1)
                                                  - moveOverhead * (2 + mtg));

    // x basetime (+ z increment)
    // If there is a healthy increment, timeLeft can exceed the actual available
    // game time for the current move, so also cap to a percentage of available game time.
    if (limits.movestogo == 0)
    {
        // Extra time according to timeLeft
        if (originalTimeAdjust < 0)
            originalTimeAdjust = 0.3285 * std::log10(timeLeft) - 0.4830;

        // Calculate time constants based on current time left.
        double logTimeInSec = std::log10(scaledTime / 1000.0);
        double optConstant  = std::min(0.00308 + 0.000319 * logTimeInSec, 0.00506);
        double maxConstant  = std::max(3.39 + 3.01 * logTimeInSec, 2.93);

        optScale = std::min(0.0122 + std::pow(ply + 2.95, 0.462) * optConstant,
                            0.213 * limits.time[us] / timeLeft)
                 * originalTimeAdjust;

        maxScale = std::min(6.64, maxConstant + ply / 12.0);
    }

    // x moves in y seconds (+ z increment)
    else
    {
        optScale = std::min((0.88 + ply / 116.4) / mtg, 0.88 * limits.time[us] / timeLeft);
        maxScale = std::min(6.3, 1.5 + 0.11 * mtg);
    }

    // Limit the maximum possible time for this move
    optimumTime = TimePoint(optScale * timeLeft);
    maximumTime =
      TimePoint(std::min(0.825 * limits.time[us] - moveOverhead, maxScale * optimumTime)) - 10;

    if (options["Ponder"])
        optimumTime += optimumTime / 4;
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

// TTEntry struct is the 10 bytes transposition table entry, defined as below:
//
// key        16 bit
// depth       8 bit
// generation  5 bit
// pv node     1 bit
// bound type  2 bit
// move       16 bit
// value      16 bit
// evaluation 16 bit
//
// These fields are in the same order as accessed by TT::probe(), since memory is fastest sequentially.
// Equally, the store order in save() matches this order.

struct TTEntry {

    // Convert internal bitfields to external types
    TTData read() const {
        return TTData{Move(move16),           Value(value16),
                      Value(eval16),          Depth(depth8 + DEPTH_ENTRY_OFFSET),
                      Bound(genBound8 & 0x3), bool(genBound8 & 0x4)};
    }

    bool is_occupied() const;
    void save(Key k, Value v, bool pv, Bound b, Depth d, Move m, Value ev, uint8_t generation8);
    // The returned age is a multiple of TranspositionTable::GENERATION_DELTA
    uint8_t relative_age(const uint8_t generation8) const;

   private:
    friend class TranspositionTable;

    uint16_t key16;
    uint8_t  depth8;
    uint8_t  genBound8;
    Move     move16;
    int16_t  value16;
    int16_t  eval16;
};

// `genBound8` is where most of the details are. We use the following constants to manipulate 5 leading generation bits
// and 3 trailing miscellaneous bits.

// These bits are reserved for other things.
static constexpr unsigned GENERATION_BITS = 3;
// increment for generation field
static constexpr int GENERATION_DELTA = (1 << GENERATION_BITS);
// cycle length
static constexpr int GENERATION_CYCLE = 255 + GENERATION_DELTA;
// mask to pull out generation number
static constexpr int GENERATION_MASK = (0xFF << GENERATION_BITS) & 0xFF;

// DEPTH_ENTRY_OFFSET exists because 1) we use `bool(depth8)` as the occupancy check, but
// 2) we need to store negative depths for QS. (`depth8` is the only field with "spare bits":
// we sacrifice the ability to store depths greater than 1<<8 less the offset, as asserted in `save`.)
bool TTEntry::is_occupied() const { return bool(depth8); }

// Populates the TTEntry with a new node's data, possibly
// overwriting an old position. The update is not atomic and can be racy.
void TTEntry::save(
  Key k, Value v, bool pv, Bound b, Depth d, Move m, Value ev, uint8_t generation8) {

    // Preserve the old ttmove if we don't have a new one
    if (m || uint16_t(k) != key16)
        move16 = m;

    // Overwrite less valuable entries (cheapest checks first)
    if (b == BOUND_EXACT || uint16_t(k) != key16 || d - DEPTH_ENTRY_OFFSET + 2 * pv > depth8 - 4
        || relative_age(generation8))
    {
        assert(d > DEPTH_ENTRY_OFFSET);
        assert(d < 256 + DEPTH_ENTRY_OFFSET);

        key16     = uint16_t(k);
        depth8    = uint8_t(d - DEPTH_ENTRY_OFFSET);
        genBound8 = uint8_t(generation8 | uint8_t(pv) << 2 | b);
        value16   = int16_t(v);
        eval16    = int16_t(ev);
    }
}

uint8_t TTEntry::relative_age(const uint8_t generation8) const {
    // Due to our packed storage format for generation and its cyclic
    // nature we add GENERATION_CYCLE (256 is the modulus, plus what
    // is needed to keep the unrelated lowest n bits from affecting
    // the result) to calculate the entry age correctly even after
    // generation8 overflows into the next cycle.
    return (GENERATION_CYCLE + generation8 - genBound8) & GENERATION_MASK;
}

// TTWriter is but a very thin wrapper around the pointer
TTWriter::TTWriter(TTEntry* tte) :
    entry(tte) {}

void TTWriter::write(
  Key k, Value v, bool pv, Bound b, Depth d, Move m, Value ev, uint8_t generation8) {
    entry->save(k, v, pv, b, d, m, ev, generation8);
}

// A TranspositionTable is an array of Cluster, of size clusterCount. Each cluster consists of ClusterSize number
// of TTEntry. Each non-empty TTEntry contains information on exactly one position. The size of a Cluster should
// divide the size of a cache line for best performance, as the cacheline is prefetched when possible.

static constexpr int ClusterSize = 3;

struct Cluster {
    TTEntry entry[ClusterSize];
    char    padding[2];  // Pad to 32 bytes
};

static_assert(sizeof(Cluster) == 32, "Suboptimal Cluster size");

// Sets the size of the transposition table,
// measured in megabytes. Transposition table consists
// of clusters and each cluster consists of ClusterSize number of TTEntry.
void TranspositionTable::resize(size_t mbSize, ThreadPool& threads) {
    aligned_large_pages_free(table);

    clusterCount = mbSize * 1024 * 1024 / sizeof(Cluster);

    table = static_cast<Cluster*>(aligned_large_pages_alloc(clusterCount * sizeof(Cluster)));

    if (!table)
    {
        std::cerr << "Failed to allocate " << mbSize << "MB for transposition table." << std::endl;
        exit(EXIT_FAILURE);
    }

    clear(threads);
}

// Initializes the entire transposition table to zero,
// in a multi-threaded way.
void TranspositionTable::clear(ThreadPool& threads) {
    generation8              = 0;
    const size_t threadCount = threads.num_threads();

    for (size_t i = 0; i < threadCount; ++i)
    {
        threads.run_on_thread(i, [this, i, threadCount]() {
            // Each thread will zero its part of the hash table
            const size_t stride = clusterCount / threadCount;
            const size_t start  = stride * i;
            const size_t len    = i + 1 != threadCount ? stride : clusterCount - start;

            std::memset(&table[start], 0, len * sizeof(Cluster));
        });
    }

    for (size_t i = 0; i < threadCount; ++i)
        threads.wait_on_thread(i);
}

// Returns an approximation of the hashtable
// occupation during a search. The hash is x permill full, as per UCI protocol.
// Only counts entries which match the current generation.
int TranspositionTable::hashfull() const {

    int cnt = 0;
    for (int i = 0; i < 1000; ++i)
        for (int j = 0; j < ClusterSize; ++j)
            cnt += table[i].entry[j].is_occupied()
                && (table[i].entry[j].genBound8 & GENERATION_MASK) == generation8;

    return cnt / ClusterSize;
}

void TranspositionTable::new_search() {
    // increment by delta to keep lower bits as is
    generation8 += GENERATION_DELTA;
}

uint8_t TranspositionTable::generation() const { return generation8; }

// Looks up the current position in the transposition
// table. It returns true if the position is found.
// Otherwise, it returns false and a pointer to an empty or least valuable TTEntry
// to be replaced later. The replace value of an entry is calculated as its depth
// minus 8 times its relative age. TTEntry t1 is considered more valuable than
// TTEntry t2 if its replace value is greater than that of t2.
std::tuple<bool, TTData, TTWriter> TranspositionTable::probe(const Key key) const {

    TTEntry* const tte   = first_entry(key);
    const uint16_t key16 = uint16_t(key);  // Use the low 16 bits as key inside the cluster

    for (int i = 0; i < ClusterSize; ++i)
        if (tte[i].key16 == key16)
            // This gap is the main place for read races.
            // After `read()` completes that copy is final, but may be self-inconsistent.
            return {tte[i].is_occupied(), tte[i].read(), TTWriter(&tte[i])};

    // Find an entry to be replaced according to the replacement strategy
    TTEntry* replace = tte;
    for (int i = 1; i < ClusterSize; ++i)
        if (replace->depth8 - replace->relative_age(generation8) * 2
            > tte[i].depth8 - tte[i].relative_age(generation8) * 2)
            replace = &tte[i];

    return {false, TTData(), TTWriter(replace)};
}

TTEntry* TranspositionTable::first_entry(const Key key) const {
    return &table[mul_hi64(key, clusterCount)].entry[0];
}

}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

bool CaseInsensitiveLess::operator()(const std::string& s1, const std::string& s2) const {

    return std::lexicographical_compare(
      s1.begin(), s1.end(), s2.begin(), s2.end(),
      [](char c1, char c2) { return std::tolower(c1) < std::tolower(c2); });
}

void OptionsMap::add_info_listener(InfoListener&& message_func) { info = std::move(message_func); }

void OptionsMap::setoption(std::istringstream& is) {
    std::string token, name, value;

    is >> token;  // Consume the "name" token

    // Read the option name (can contain spaces)
    while (is >> token && token != "value")
        name += (name.empty() ? "" : " ") + token;

    // Read the option value (can contain spaces)
    while (is >> token)
        value += (value.empty() ? "" : " ") + token;

    if (options_map.count(name))
        options_map[name] = value;
    else
        sync_cout << "No such option: " << name << sync_endl;
}

Option OptionsMap::operator[](const std::string& name) const {
    auto it = options_map.find(name);
    return it != options_map.end() ? it->second : Option(this);
}

Option& OptionsMap::operator[](const std::string& name) {
    if (!options_map.count(name))
        options_map[name] = Option(this);
    return options_map[name];
}

std::size_t OptionsMap::count(const std::string& name) const { return options_map.count(name); }

Option::Option(const OptionsMap* map) :
    parent(map) {}

Option::Option(const char* v, OnChange f) :
    type("string"),
    min(0),
    max(0),
    on_change(std::move(f)) {
    defaultValue = currentValue = v;
}

Option::Option(bool v, OnChange f) :
    type("check"),
    min(0),
    max(0),
    on_change(std::move(f)) {
    defaultValue = currentValue = (v ? "true" : "false");
}

Option::Option(OnChange f) :
    type("button"),
    min(0),
    max(0),
    on_change(std::move(f)) {}

Option::Option(double v, int minv, int maxv, OnChange f) :
    type("spin"),
    min(minv),
    max(maxv),
    on_change(std::move(f)) {
    defaultValue = currentValue = std::to_string(v);
}

Option::Option(const char* v, const char* cur, OnChange f) :
    type("combo"),
    min(0),
    max(0),
    on_change(std::move(f)) {
    defaultValue = v;
    currentValue = cur;
}

Option::operator int() const {
    assert(type == "check" || type == "spin");
    return (type == "spin" ? std::stoi(currentValue) : currentValue == "true");
}

Option::operator std::string() const {
    assert(type == "string");
    return currentValue;
}

bool Option::operator==(const char* s) const {
    assert(type == "combo");
    return !CaseInsensitiveLess()(currentValue, s) && !CaseInsensitiveLess()(s, currentValue);
}

bool Option::operator!=(const char* s) const { return !(*this == s); }

// Inits options and assigns idx in the correct printing order

void Option::operator<<(const Option& o) {

    static size_t insert_order = 0;

    auto p = this->parent;
    *this  = o;

    this->parent = p;
    idx          = insert_order++;
}

// Updates currentValue and triggers on_change() action. It's up to
// the GUI to check for option's limits, but we could receive the new value
// from the user by console window, so let's check the bounds anyway.
Option& Option::operator=(const std::string& v) {

    assert(!type.empty());

    if ((type != "button" && type != "string" && v.empty())
        || (type == "check" && v != "true" && v != "false")
        || (type == "spin" && (std::stof(v) < min || std::stof(v) > max)))
        return *this;

    if (type == "combo")
    {
        OptionsMap         comboMap;  // To have case insensitive compare
        std::string        token;
        std::istringstream ss(defaultValue);
        while (ss >> token)
            comboMap[token] << Option();
        if (!comboMap.count(v) || v == "var")
            return *this;
    }

    if (type == "string")
        currentValue = v == "<empty>" ? "" : v;
    else if (type != "button")
        currentValue = v;

    if (on_change)
    {
        const auto ret = on_change(*this);

        if (ret && parent != nullptr && parent->info != nullptr)
            parent->info(ret);
    }

    return *this;
}

std::ostream& operator<<(std::ostream& os, const OptionsMap& om) {
    for (size_t idx = 0; idx < om.options_map.size(); ++idx)
        for (const auto& it : om.options_map)
            if (it.second.idx == idx)
            {
                const Option& o = it.second;
                os << "\noption name " << it.first << " type " << o.type;

                if (o.type == "check" || o.type == "combo")
                    os << " default " << o.defaultValue;

                else if (o.type == "string")
                {
                    std::string defaultValue = o.defaultValue.empty() ? "<empty>" : o.defaultValue;
                    os << " default " << defaultValue;
                }

                else if (o.type == "spin")
                    os << " default " << int(stof(o.defaultValue)) << " min " << o.min << " max "
                       << o.max;

                break;
            }

    return os;
}
}

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

template<typename... Ts>
struct overload: Ts... {
    using Ts::operator()...;
};

template<typename... Ts>
overload(Ts...) -> overload<Ts...>;

void UCIEngine::print_info_string(std::string_view str) {
    sync_cout_start();
    for (auto& line : split(str, "\n"))
    {
        if (!is_whitespace(line))
        {
            std::cout << "info string " << line << '\n';
        }
    }
    sync_cout_end();
}

UCIEngine::UCIEngine(int argc, char** argv) :
    engine("GreenChess"),
    cli(argc, argv) {

    engine.get_options().add_info_listener([](const std::optional<std::string>& str) {
        if (str.has_value())
            print_info_string(*str);
    });

    engine.set_on_iter([](const auto& i) { on_iter(i); });
    engine.set_on_update_no_moves([](const auto& i) { on_update_no_moves(i); });
    engine.set_on_update_full(
      [this](const auto& i) { on_update_full(i, engine.get_options()["UCI_ShowWDL"]); });
    engine.set_on_bestmove([](const auto& bm, const auto& p) { on_bestmove(bm, p); });
}

void UCIEngine::loop() {
    std::string token, cmd;

    for (int i = 1; i < cli.argc; ++i)
        cmd += std::string(cli.argv[i]) + " ";

    do
    {
        if (cli.argc == 1
            && !getline(std::cin, cmd))  // Wait for an input or an end-of-file (EOF) indication
            cmd = "quit";

        std::istringstream is(cmd);

        token.clear();  // Avoid a stale if getline() returns nothing or a blank line
        is >> std::skipws >> token;

        if (token == "quit" || token == "stop")
            engine.stop();

        // The GUI sends 'ponderhit' to tell that the user has played the expected move.
        // So, 'ponderhit' is sent if pondering was done on the same move that the user
        // has played. The search should continue, but should also switch from pondering
        // to the normal search.
        else if (token == "ponderhit")
            engine.set_ponderhit(false);

        else if (token == "uci")
        {
            sync_cout << "id name " << engine_info(true) << "\n"
                      << engine.get_options() << sync_endl;

            sync_cout << "uciok" << sync_endl;
        }

        else if (token == "setoption")
            setoption(is);
        else if (token == "go")
        {
            // send info strings after the go command is sent for old GUIs and python-chess
            print_info_string(engine.numa_config_information_as_string());
            print_info_string(engine.thread_binding_information_as_string());
            go(is);
        }
        else if (token == "position")
            position(is);
        else if (token == "ucinewgame")
            engine.search_clear();
        else if (token == "isready")
            sync_cout << "readyok" << sync_endl;

        // Add custom non-UCI commands, mainly for debugging purposes.
        // These commands must not be used during a search!
        else if (token == "flip")
            engine.flip();
        else if (token == "bench")
            bench(is);
        else if (token == "d")
            sync_cout << engine.visualize() << sync_endl;
        else if (token == "eval")
            engine.trace_eval();
        else if (token == "compiler")
            sync_cout << compiler_info() << sync_endl;
        else if (token == "export_net")
        {
            std::pair<std::optional<std::string>, std::string> files[2];

            if (is >> std::skipws >> files[0].second)
                files[0].first = files[0].second;

            if (is >> std::skipws >> files[1].second)
                files[1].first = files[1].second;

            engine.save_network(files);
        }
        else if (token == "--help" || token == "help" || token == "--license" || token == "license")
            sync_cout
              << "\nStockfish is a powerful chess engine for playing and analyzing."
                 "\nIt is released as free software licensed under the GNU GPLv3 License."
                 "\nStockfish is normally used with a graphical user interface (GUI) and implements"
                 "\nthe Universal Chess Interface (UCI) protocol to communicate with a GUI, an API, etc."
                 "\nFor any further information, visit https://github.com/official-stockfish/Stockfish#readme"
                 "\nor read the corresponding README.md and Copying.txt files distributed along with this program.\n"
              << sync_endl;
        else if (!token.empty() && token[0] != '#')
            sync_cout << "Unknown command: '" << cmd << "'. Type help for more information."
                      << sync_endl;

    } while (token != "quit" && cli.argc == 1);  // The command-line arguments are one-shot
}

Search::LimitsType UCIEngine::parse_limits(std::istream& is) {
    Search::LimitsType limits;
    std::string        token;

    limits.startTime = now();  // The search starts as early as possible

    while (is >> token)
        if (token == "searchmoves")  // Needs to be the last command on the line
            while (is >> token)
                limits.searchmoves.push_back(to_lower(token));

        else if (token == "wtime")
            is >> limits.time[WHITE];
        else if (token == "btime")
            is >> limits.time[BLACK];
        else if (token == "winc")
            is >> limits.inc[WHITE];
        else if (token == "binc")
            is >> limits.inc[BLACK];
        else if (token == "movestogo")
            is >> limits.movestogo;
        else if (token == "depth")
            is >> limits.depth;
        else if (token == "nodes")
            is >> limits.nodes;
        else if (token == "movetime")
            is >> limits.movetime;
        else if (token == "mate")
            is >> limits.mate;
        else if (token == "perft")
            is >> limits.perft;
        else if (token == "infinite")
            limits.infinite = 1;
        else if (token == "ponder")
            limits.ponderMode = true;

    return limits;
}

void UCIEngine::go(std::istringstream& is) {

    Search::LimitsType limits = parse_limits(is);

    if (limits.perft)
        perft(limits);
    else
        engine.go(limits);
}

void UCIEngine::bench(std::istream& args) {
    std::string token;
    uint64_t    num, nodes = 0, cnt = 1;
    uint64_t    nodesSearched = 0;
    const auto& options       = engine.get_options();

    engine.set_on_update_full([&](const auto& i) {
        nodesSearched = i.nodes;
        on_update_full(i, options["UCI_ShowWDL"]);
    });

    std::vector<std::string> list = Benchmark::setup_bench(engine.fen(), args);

    num = count_if(list.begin(), list.end(),
                   [](const std::string& s) { return s.find("go ") == 0 || s.find("eval") == 0; });

    TimePoint elapsed = now();

    for (const auto& cmd : list)
    {
        std::istringstream is(cmd);
        is >> std::skipws >> token;

        if (token == "go" || token == "eval")
        {
            std::cerr << "\nPosition: " << cnt++ << '/' << num << " (" << engine.fen() << ")"
                      << std::endl;
            if (token == "go")
            {
                Search::LimitsType limits = parse_limits(is);

                if (limits.perft)
                    nodesSearched = perft(limits);
                else
                {
                    engine.go(limits);
                    engine.wait_for_search_finished();
                }

                nodes += nodesSearched;
                nodesSearched = 0;
            }
            else
                engine.trace_eval();
        }
        else if (token == "setoption")
            setoption(is);
        else if (token == "position")
            position(is);
        else if (token == "ucinewgame")
        {
            engine.search_clear();  // search_clear may take a while
            elapsed = now();
        }
    }

    elapsed = now() - elapsed + 1;  // Ensure positivity to avoid a 'divide by zero'

    dbg_print();

    std::cerr << "\n==========================="    //
              << "\nTotal time (ms) : " << elapsed  //
              << "\nNodes searched  : " << nodes    //
              << "\nNodes/second    : " << 1000 * nodes / elapsed << std::endl;

    // reset callback, to not capture a dangling reference to nodesSearched
    engine.set_on_update_full([&](const auto& i) { on_update_full(i, options["UCI_ShowWDL"]); });
}

void UCIEngine::setoption(std::istringstream& is) {
    engine.wait_for_search_finished();
    engine.get_options().setoption(is);
}

std::uint64_t UCIEngine::perft(const Search::LimitsType& limits) {
    auto nodes = engine.perft(engine.fen(), limits.perft, engine.get_options()["UCI_Chess960"]);
    sync_cout << "\nNodes searched: " << nodes << "\n" << sync_endl;
    return nodes;
}

void UCIEngine::position(std::istringstream& is) {
    std::string token, fen;

    is >> token;

    if (token == "startpos")
    {
        fen = StartFEN;
        is >> token;  // Consume the "moves" token, if any
    }
    else if (token == "fen")
        while (is >> token && token != "moves")
            fen += token + " ";
    else
        return;

    std::vector<std::string> moves;

    while (is >> token)
    {
        moves.push_back(token);
    }

    engine.set_position(fen, moves);
}

namespace {

struct WinRateParams {
    double a;
    double b;
};

WinRateParams win_rate_params(const Position& pos) {

    int material = pos.count<PAWN>() + 3 * pos.count<KNIGHT>() + 3 * pos.count<BISHOP>()
                 + 5 * pos.count<ROOK>() + 9 * pos.count<QUEEN>();

    // The fitted model only uses data for material counts in [17, 78], and is anchored at count 58.
    double m = std::clamp(material, 17, 78) / 58.0;

    // Return a = p_a(material) and b = p_b(material), see github.com/official-stockfish/WDL_model
    constexpr double as[] = {-37.45051876, 121.19101539, -132.78783573, 420.70576692};
    constexpr double bs[] = {90.26261072, -137.26549898, 71.10130540, 51.35259597};

    double a = (((as[0] * m + as[1]) * m + as[2]) * m) + as[3];
    double b = (((bs[0] * m + bs[1]) * m + bs[2]) * m) + bs[3];

    return {a, b};
}

// The win rate model is 1 / (1 + exp((a - eval) / b)), where a = p_a(material) and b = p_b(material).
// It fits the LTC fishtest statistics rather accurately.
int win_rate_model(Value v, const Position& pos) {

    auto [a, b] = win_rate_params(pos);

    // Return the win rate in per mille units, rounded to the nearest integer.
    return int(0.5 + 1000 / (1 + std::exp((a - double(v)) / b)));
}
}

std::string UCIEngine::format_score(const Score& s) {
    constexpr int TB_CP = 20000;
    const auto    format =
      overload{[](Score::Mate mate) -> std::string {
                   auto m = (mate.plies > 0 ? (mate.plies + 1) : mate.plies) / 2;
                   return std::string("mate ") + std::to_string(m);
               },
               [](Score::Tablebase tb) -> std::string {
                   return std::string("cp ")
                        + std::to_string((tb.win ? TB_CP - tb.plies : -TB_CP - tb.plies));
               },
               [](Score::InternalUnits units) -> std::string {
                   return std::string("cp ") + std::to_string(units.value);
               }};

    return s.visit(format);
}

// Turns a Value to an integer centipawn number,
// without treatment of mate and similar special scores.
int UCIEngine::to_cp(Value v, const Position& pos) {

    // In general, the score can be defined via the WDL as
    // (log(1/L - 1) - log(1/W - 1)) / (log(1/L - 1) + log(1/W - 1)).
    // Based on our win_rate_model, this simply yields v / a.

    auto [a, b] = win_rate_params(pos);

    return std::round(100 * int(v) / a);
}

std::string UCIEngine::wdl(Value v, const Position& pos) {
    std::stringstream ss;

    int wdl_w = win_rate_model(v, pos);
    int wdl_l = win_rate_model(-v, pos);
    int wdl_d = 1000 - wdl_w - wdl_l;
    ss << wdl_w << " " << wdl_d << " " << wdl_l;

    return ss.str();
}

std::string UCIEngine::square(Square s) {
    return std::string{char('a' + file_of(s)), char('1' + rank_of(s))};
}

std::string UCIEngine::move(Move m, bool chess960) {
    if (m == Move::none())
        return "(none)";

    if (m == Move::null())
        return "0000";

    Square from = m.from_sq();
    Square to   = m.to_sq();

    if (m.type_of() == CASTLING && !chess960)
        to = make_square(to > from ? FILE_G : FILE_C, rank_of(from));

    std::string move = square(from) + square(to);

    if (m.type_of() == PROMOTION)
        move += " pnbrqk"[m.promotion_type()];

    return move;
}
```

```
std::string UCIEngine::to_lower(std::string str) {
    std::transform(str.begin(), str.end(), str.begin(), [](auto c) { return std::tolower(c); });

    return str;
}

Move UCIEngine::to_move(const Position& pos, std::string str) {
    str = to_lower(str);

    for (const auto& m : MoveList<LEGAL>(pos))
        if (str == move(m, pos.is_chess960()))
            return m;

    return Move::none();
}

void UCIEngine::on_update_no_moves(const Engine::InfoShort& info) {
    sync_cout << "info depth " << info.depth << " score " << format_score(info.score) << sync_endl;
}

void UCIEngine::on_update_full(const Engine::InfoFull& info, bool showWDL) {
    std::stringstream ss;

    ss << "info";
    ss << " depth " << info.depth                 //
       << " seldepth " << info.selDepth           //
       << " multipv " << info.multiPV             //
       << " score " << format_score(info.score);  //

    if (showWDL)
        ss << " wdl " << info.wdl;

    if (!info.bound.empty())
        ss << " " << info.bound;

    ss << " nodes " << info.nodes        //
       << " nps " << info.nps            //
       << " hashfull " << info.hashfull  //
       << " tbhits " << info.tbHits      //
       << " time " << info.timeMs        //
       << " pv " << info.pv;             //

    sync_cout << ss.str() << sync_endl;
}

void UCIEngine::on_iter(const Engine::InfoIter& info) {
    std::stringstream ss;

    ss << "info";
    ss << " depth " << info.depth                     //
       << " currmove " << info.currmove               //
       << " currmovenumber " << info.currmovenumber;  //

    sync_cout << ss.str() << sync_endl;
}

void UCIEngine::on_bestmove(std::string_view bestmove, std::string_view ponder) {
    sync_cout << "bestmove " << bestmove;
    if (!ponder.empty())
        std::cout << " ponder " << ponder;
    std::cout << sync_endl;
}

}  // namespace Stockfish
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#if __has_include("features.h")
#endif

#if defined(__linux__) && !defined(__ANDROID__)
#endif

#if defined(__APPLE__) || defined(__ANDROID__) || defined(__OpenBSD__) \
  || (defined(__GLIBCXX__) && !defined(_GLIBCXX_HAVE_ALIGNED_ALLOC) && !defined(_WIN32)) \
  || defined(__e2k__)
    #define POSIXALIGNEDALLOC
#endif

#ifdef _WIN32
    #if _WIN32_WINNT < 0x0601
        #undef _WIN32_WINNT
        #define _WIN32_WINNT 0x0601  // Force to include needed API prototypes
    #endif

    #ifndef NOMINMAX
        #define NOMINMAX
    #endif

// The needed Windows API for processor groups could be missed from old Windows
// versions, so instead of calling them directly (forcing the linker to resolve
// the calls at compile time), try to load them at runtime. To do this we need
// first to define the corresponding function pointers.

extern "C" {
using OpenProcessToken_t      = bool (*)(HANDLE, DWORD, PHANDLE);
using LookupPrivilegeValueA_t = bool (*)(LPCSTR, LPCSTR, PLUID);
using AdjustTokenPrivileges_t =
  bool (*)(HANDLE, BOOL, PTOKEN_PRIVILEGES, DWORD, PTOKEN_PRIVILEGES, PDWORD);
}
#endif

namespace Stockfish {

// Wrappers for systems where the c++17 implementation does not guarantee the
// availability of aligned_alloc(). Memory allocated with std_aligned_alloc()
// must be freed with std_aligned_free().

void* std_aligned_alloc(size_t alignment, size_t size) {
#if defined(_ISOC11_SOURCE)
    return aligned_alloc(alignment, size);
#elif defined(POSIXALIGNEDALLOC)
    void* mem = nullptr;
    posix_memalign(&mem, alignment, size);
    return mem;
#elif defined(_WIN32) && !defined(_M_ARM) && !defined(_M_ARM64)
    return _mm_malloc(size, alignment);
#elif defined(_WIN32)
    return _aligned_malloc(size, alignment);
#else
    return std::aligned_alloc(alignment, size);
#endif
}

void std_aligned_free(void* ptr) {

#if defined(POSIXALIGNEDALLOC)
    free(ptr);
#elif defined(_WIN32) && !defined(_M_ARM) && !defined(_M_ARM64)
    _mm_free(ptr);
#elif defined(_WIN32)
    _aligned_free(ptr);
#else
    free(ptr);
#endif
}

// aligned_large_pages_alloc() will return suitably aligned memory,
// if possible using large pages.

#if defined(_WIN32)

static void* aligned_large_pages_alloc_windows([[maybe_unused]] size_t allocSize) {

    #if !defined(_WIN64)
    return nullptr;
    #else

    HANDLE hProcessToken{};
    LUID   luid{};
    void*  mem = nullptr;

    const size_t largePageSize = GetLargePageMinimum();
    if (!largePageSize)
        return nullptr;

    // Dynamically link OpenProcessToken, LookupPrivilegeValue and AdjustTokenPrivileges

    HMODULE hAdvapi32 = GetModuleHandle(TEXT("advapi32.dll"));

    if (!hAdvapi32)
        hAdvapi32 = LoadLibrary(TEXT("advapi32.dll"));

    auto OpenProcessToken_f =
      OpenProcessToken_t((void (*)()) GetProcAddress(hAdvapi32, "OpenProcessToken"));
    if (!OpenProcessToken_f)
        return nullptr;
    auto LookupPrivilegeValueA_f =
      LookupPrivilegeValueA_t((void (*)()) GetProcAddress(hAdvapi32, "LookupPrivilegeValueA"));
    if (!LookupPrivilegeValueA_f)
        return nullptr;
    auto AdjustTokenPrivileges_f =
      AdjustTokenPrivileges_t((void (*)()) GetProcAddress(hAdvapi32, "AdjustTokenPrivileges"));
    if (!AdjustTokenPrivileges_f)
        return nullptr;

    // We need SeLockMemoryPrivilege, so try to enable it for the process

    if (!OpenProcessToken_f(  // OpenProcessToken()
          GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hProcessToken))
        return nullptr;

    if (LookupPrivilegeValueA_f(nullptr, "SeLockMemoryPrivilege", &luid))
    {
        TOKEN_PRIVILEGES tp{};
        TOKEN_PRIVILEGES prevTp{};
        DWORD            prevTpLen = 0;

        tp.PrivilegeCount           = 1;
        tp.Privileges[0].Luid       = luid;
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

        // Try to enable SeLockMemoryPrivilege. Note that even if AdjustTokenPrivileges()
        // succeeds, we still need to query GetLastError() to ensure that the privileges
        // were actually obtained.

        if (AdjustTokenPrivileges_f(hProcessToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES), &prevTp,
                                    &prevTpLen)
            && GetLastError() == ERROR_SUCCESS)
        {
            // Round up size to full pages and allocate
            allocSize = (allocSize + largePageSize - 1) & ~size_t(largePageSize - 1);
            mem       = VirtualAlloc(nullptr, allocSize, MEM_RESERVE | MEM_COMMIT | MEM_LARGE_PAGES,
                                     PAGE_READWRITE);

            // Privilege no longer needed, restore previous state
            AdjustTokenPrivileges_f(hProcessToken, FALSE, &prevTp, 0, nullptr, nullptr);
        }
    }

    CloseHandle(hProcessToken);

    return mem;

    #endif
}

void* aligned_large_pages_alloc(size_t allocSize) {

    // Try to allocate large pages
    void* mem = aligned_large_pages_alloc_windows(allocSize);

    // Fall back to regular, page-aligned, allocation if necessary
    if (!mem)
        mem = VirtualAlloc(nullptr, allocSize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

    return mem;
}

#else

void* aligned_large_pages_alloc(size_t allocSize) {

    #if defined(__linux__)
    constexpr size_t alignment = 2 * 1024 * 1024;  // 2MB page size assumed
    #else
    constexpr size_t alignment = 4096;  // small page size assumed
    #endif

    // Round up to multiples of alignment
    size_t size = ((allocSize + alignment - 1) / alignment) * alignment;
    void*  mem  = std_aligned_alloc(alignment, size);
    #if defined(MADV_HUGEPAGE)
    madvise(mem, size, MADV_HUGEPAGE);
    #endif
    return mem;
}

#endif

// aligned_large_pages_free() will free the previously memory allocated
// by aligned_large_pages_alloc(). The effect is a nop if mem == nullptr.

#if defined(_WIN32)

void aligned_large_pages_free(void* mem) {

    if (mem && !VirtualFree(mem, 0, MEM_RELEASE))
    {
        DWORD err = GetLastError();
        std::cerr << "Failed to free large page memory. Error code: 0x" << std::hex << err
                  << std::dec << std::endl;
        exit(EXIT_FAILURE);
    }
}

#else

void aligned_large_pages_free(void* mem) { std_aligned_free(mem); }

#endif
}  // namespace Stockfish

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace Stockfish {

Score::Score(Value v, const Position& pos) {
    assert(-VALUE_INFINITE < v && v < VALUE_INFINITE);

    if (std::abs(v) < VALUE_TB_WIN_IN_MAX_PLY)
    {
        score = InternalUnits{UCIEngine::to_cp(v, pos)};
    }
    else if (std::abs(v) <= VALUE_TB)
    {
        auto distance = VALUE_TB - std::abs(v);
        score         = (v > 0) ? Tablebase{distance, true} : Tablebase{-distance, false};
    }
    else
    {
        auto distance = VALUE_MATE - std::abs(v);
        score         = (v > 0) ? Mate{distance} : Mate{-distance};
    }
}

}
/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

namespace {
// Macro to embed the default efficiently updatable neural network (NNUE) file
// data in the engine binary (using incbin.h, by Dale Weiler).
// This macro invocation will declare the following three variables
//     const unsigned char        gEmbeddedNNUEData[];  // a pointer to the embedded data
//     const unsigned char *const gEmbeddedNNUEEnd;     // a marker to the end
//     const unsigned int         gEmbeddedNNUESize;    // the size of the embedded file
// Note that this does not work in Microsoft Visual Studio.
#if !defined(_MSC_VER) && !defined(NNUE_EMBEDDING_OFF)
INCBIN(EmbeddedNNUEBig, EvalFileDefaultNameBig);
INCBIN(EmbeddedNNUESmall, EvalFileDefaultNameSmall);
#else
const unsigned char        gEmbeddedNNUEBigData[1]   = {0x0};
const unsigned char* const gEmbeddedNNUEBigEnd       = &gEmbeddedNNUEBigData[1];
const unsigned int         gEmbeddedNNUEBigSize      = 1;
const unsigned char        gEmbeddedNNUESmallData[1] = {0x0};
const unsigned char* const gEmbeddedNNUESmallEnd     = &gEmbeddedNNUESmallData[1];
const unsigned int         gEmbeddedNNUESmallSize    = 1;
#endif

struct EmbeddedNNUE {
    EmbeddedNNUE(const unsigned char* embeddedData,
                 const unsigned char* embeddedEnd,
                 const unsigned int   embeddedSize) :
        data(embeddedData),
        end(embeddedEnd),
        size(embeddedSize) {}
    const unsigned char* data;
    const unsigned char* end;
    const unsigned int   size;
};

using namespace Stockfish::Eval::NNUE;

EmbeddedNNUE get_embedded(EmbeddedNNUEType type) {
    if (type == EmbeddedNNUEType::BIG)
        return EmbeddedNNUE(gEmbeddedNNUEBigData, gEmbeddedNNUEBigEnd, gEmbeddedNNUEBigSize);
    else
        return EmbeddedNNUE(gEmbeddedNNUESmallData, gEmbeddedNNUESmallEnd, gEmbeddedNNUESmallSize);
}

}

namespace Stockfish::Eval::NNUE {

namespace Detail {

// Read evaluation function parameters
template<typename T>
bool read_parameters(std::istream& stream, T& reference) {

    std::uint32_t header;
    header = read_little_endian<std::uint32_t>(stream);
    if (!stream || header != T::get_hash_value())
        return false;
    return reference.read_parameters(stream);
}

// Write evaluation function parameters
template<typename T>
bool write_parameters(std::ostream& stream, T& reference) {

    write_little_endian<std::uint32_t>(stream, T::get_hash_value());
    return reference.write_parameters(stream);
}

}  // namespace Detail

template<typename Arch, typename Transformer>
Network<Arch, Transformer>::Network(const Network<Arch, Transformer>& other) :
    evalFile(other.evalFile),
    embeddedType(other.embeddedType) {

    if (other.featureTransformer)
        featureTransformer = make_unique_large_page<Transformer>(*other.featureTransformer);

    network = make_unique_aligned<Arch[]>(LayerStacks);

    if (!other.network)
        return;

    for (std::size_t i = 0; i < LayerStacks; ++i)
        network[i] = other.network[i];
}

template<typename Arch, typename Transformer>
Network<Arch, Transformer>&
Network<Arch, Transformer>::operator=(const Network<Arch, Transformer>& other) {
    evalFile     = other.evalFile;
    embeddedType = other.embeddedType;

    if (other.featureTransformer)
        featureTransformer = make_unique_large_page<Transformer>(*other.featureTransformer);

    network = make_unique_aligned<Arch[]>(LayerStacks);

    if (!other.network)
        return *this;

    for (std::size_t i = 0; i < LayerStacks; ++i)
        network[i] = other.network[i];

    return *this;
}

template<typename Arch, typename Transformer>
void Network<Arch, Transformer>::load(const std::string& rootDirectory, std::string evalfilePath) {
#if defined(DEFAULT_NNUE_DIRECTORY)
    std::vector<std::string> dirs = {"<internal>", "", rootDirectory,
                                     stringify(DEFAULT_NNUE_DIRECTORY)};
#else
    std::vector<std::string> dirs = {"<internal>", "", rootDirectory};
#endif

    if (evalfilePath.empty())
        evalfilePath = evalFile.defaultName;

    for (const auto& directory : dirs)
    {
        if (evalFile.current != evalfilePath)
        {
            if (directory != "<internal>")
            {
                load_user_net(directory, evalfilePath);
            }

            if (directory == "<internal>" && evalfilePath == evalFile.defaultName)
            {
                load_internal();
            }
        }
    }
}

template<typename Arch, typename Transformer>
bool Network<Arch, Transformer>::save(const std::optional<std::string>& filename) const {
    std::string actualFilename;
    std::string msg;

    if (filename.has_value())
        actualFilename = filename.value();
    else
    {
        if (evalFile.current != evalFile.defaultName)
        {
            msg = "Failed to export a net. "
                  "A non-embedded net can only be saved if the filename is specified";

            sync_cout << msg << sync_endl;
            return false;
        }

        actualFilename = evalFile.defaultName;
    }

    std::ofstream stream(actualFilename, std::ios_base::binary);
    bool          saved = save(stream, evalFile.current, evalFile.netDescription);

    msg = saved ? "Network saved successfully to " + actualFilename : "Failed to export a net";

    sync_cout << msg << sync_endl;
    return saved;
}

template<typename Arch, typename Transformer>
NetworkOutput
Network<Arch, Transformer>::evaluate(const Position&                         pos,
                                     AccumulatorCaches::Cache<FTDimensions>* cache) const {
    // We manually align the arrays on the stack because with gcc < 9.3
    // overaligning stack variables with alignas() doesn't work correctly.

    constexpr uint64_t alignment = CacheLineSize;

#if defined(ALIGNAS_ON_STACK_VARIABLES_BROKEN)
    TransformedFeatureType
      transformedFeaturesUnaligned[FeatureTransformer<FTDimensions, nullptr>::BufferSize
                                   + alignment / sizeof(TransformedFeatureType)];

    auto* transformedFeatures = align_ptr_up<alignment>(&transformedFeaturesUnaligned[0]);
#else
    alignas(alignment) TransformedFeatureType
      transformedFeatures[FeatureTransformer<FTDimensions, nullptr>::BufferSize];
#endif

    ASSERT_ALIGNED(transformedFeatures, alignment);

    const int  bucket     = (pos.count<ALL_PIECES>() - 1) / 4;
    const auto psqt       = featureTransformer->transform(pos, cache, transformedFeatures, bucket);
    const auto positional = network[bucket].propagate(transformedFeatures);
    return {static_cast<Value>(psqt / OutputScale), static_cast<Value>(positional / OutputScale)};
}

template<typename Arch, typename Transformer>
void Network<Arch, Transformer>::verify(std::string evalfilePath) const {
    if (evalfilePath.empty())
        evalfilePath = evalFile.defaultName;

    if (evalFile.current != evalfilePath)
    {
        std::string msg1 =
          "Network evaluation parameters compatible with the engine must be available.";
        std::string msg2 = "The network file " + evalfilePath + " was not loaded successfully.";
        std::string msg3 = "The UCI option EvalFile might need to specify the full path, "
                           "including the directory name, to the network file.";
        std::string msg4 = "The default net can be downloaded from: "
                           "https://tests.stockfishchess.org/api/nn/"
                           + evalFile.defaultName;
        std::string msg5 = "The engine will be terminated now.";

        sync_cout << "info string ERROR: " << msg1 << sync_endl;
        sync_cout << "info string ERROR: " << msg2 << sync_endl;
        sync_cout << "info string ERROR: " << msg3 << sync_endl;
        sync_cout << "info string ERROR: " << msg4 << sync_endl;
        sync_cout << "info string ERROR: " << msg5 << sync_endl;
        exit(EXIT_FAILURE);
    }

    size_t size = sizeof(*featureTransformer) + sizeof(Arch) * LayerStacks;
    sync_cout << "info string NNUE evaluation using " << evalfilePath << " ("
              << size / (1024 * 1024) << "MiB, (" << featureTransformer->InputDimensions << ", "
              << network[0].TransformedFeatureDimensions << ", " << network[0].FC_0_OUTPUTS << ", "
              << network[0].FC_1_OUTPUTS << ", 1))" << sync_endl;
}

template<typename Arch, typename Transformer>
void Network<Arch, Transformer>::hint_common_access(
  const Position& pos, AccumulatorCaches::Cache<FTDimensions>* cache) const {
    featureTransformer->hint_common_access(pos, cache);
}

template<typename Arch, typename Transformer>
NnueEvalTrace
Network<Arch, Transformer>::trace_evaluate(const Position&                         pos,
                                           AccumulatorCaches::Cache<FTDimensions>* cache) const {
    // We manually align the arrays on the stack because with gcc < 9.3
    // overaligning stack variables with alignas() doesn't work correctly.
    constexpr uint64_t alignment = CacheLineSize;

#if defined(ALIGNAS_ON_STACK_VARIABLES_BROKEN)
    TransformedFeatureType
      transformedFeaturesUnaligned[FeatureTransformer<FTDimensions, nullptr>::BufferSize
                                   + alignment / sizeof(TransformedFeatureType)];

    auto* transformedFeatures = align_ptr_up<alignment>(&transformedFeaturesUnaligned[0]);
#else
    alignas(alignment) TransformedFeatureType
      transformedFeatures[FeatureTransformer<FTDimensions, nullptr>::BufferSize];
#endif

    ASSERT_ALIGNED(transformedFeatures, alignment);

    NnueEvalTrace t{};
    t.correctBucket = (pos.count<ALL_PIECES>() - 1) / 4;
    for (IndexType bucket = 0; bucket < LayerStacks; ++bucket)
    {
        const auto materialist =
          featureTransformer->transform(pos, cache, transformedFeatures, bucket);
        const auto positional = network[bucket].propagate(transformedFeatures);

        t.psqt[bucket]       = static_cast<Value>(materialist / OutputScale);
        t.positional[bucket] = static_cast<Value>(positional / OutputScale);
    }

    return t;
}

template<typename Arch, typename Transformer>
void Network<Arch, Transformer>::load_user_net(const std::string& dir,
                                               const std::string& evalfilePath) {
    std::ifstream stream(dir + evalfilePath, std::ios::binary);
    auto          description = load(stream);

    if (description.has_value())
    {
        evalFile.current        = evalfilePath;
        evalFile.netDescription = description.value();
    }
}

template<typename Arch, typename Transformer>
void Network<Arch, Transformer>::load_internal() {
    // C++ way to prepare a buffer for a memory stream
    class MemoryBuffer: public std::basic_streambuf<char> {
       public:
        MemoryBuffer(char* p, size_t n) {
            setg(p, p, p + n);
            setp(p, p + n);
        }
    };

    const auto embedded = get_embedded(embeddedType);

    MemoryBuffer buffer(const_cast<char*>(reinterpret_cast<const char*>(embedded.data)),
                        size_t(embedded.size));

    std::istream stream(&buffer);
    auto         description = load(stream);

    if (description.has_value())
    {
        evalFile.current        = evalFile.defaultName;
        evalFile.netDescription = description.value();
    }
}

template<typename Arch, typename Transformer>
void Network<Arch, Transformer>::initialize() {
    featureTransformer = make_unique_large_page<Transformer>();
    network            = make_unique_aligned<Arch[]>(LayerStacks);
}

template<typename Arch, typename Transformer>
bool Network<Arch, Transformer>::save(std::ostream&      stream,
                                      const std::string& name,
                                      const std::string& netDescription) const {
    if (name.empty() || name == "None")
        return false;

    return write_parameters(stream, netDescription);
}

template<typename Arch, typename Transformer>
std::optional<std::string> Network<Arch, Transformer>::load(std::istream& stream) {
    initialize();
    std::string description;

    return read_parameters(stream, description) ? std::make_optional(description) : std::nullopt;
}

// Read network header
template<typename Arch, typename Transformer>
bool Network<Arch, Transformer>::read_header(std::istream&  stream,
                                             std::uint32_t* hashValue,
                                             std::string*   desc) const {
    std::uint32_t version, size;

    version    = read_little_endian<std::uint32_t>(stream);
    *hashValue = read_little_endian<std::uint32_t>(stream);
    size       = read_little_endian<std::uint32_t>(stream);
    if (!stream || version != Version)
        return false;
    desc->resize(size);
    stream.read(&(*desc)[0], size);
    return !stream.fail();
}

// Write network header
template<typename Arch, typename Transformer>
bool Network<Arch, Transformer>::write_header(std::ostream&      stream,
                                              std::uint32_t      hashValue,
                                              const std::string& desc) const {
    write_little_endian<std::uint32_t>(stream, Version);
    write_little_endian<std::uint32_t>(stream, hashValue);
    write_little_endian<std::uint32_t>(stream, std::uint32_t(desc.size()));
    stream.write(&desc[0], desc.size());
    return !stream.fail();
}

template<typename Arch, typename Transformer>
bool Network<Arch, Transformer>::read_parameters(std::istream& stream,
                                                 std::string&  netDescription) const {
    std::uint32_t hashValue;
    if (!read_header(stream, &hashValue, &netDescription))
        return false;
    if (hashValue != Network::hash)
        return false;
    if (!Detail::read_parameters(stream, *featureTransformer))
        return false;
    for (std::size_t i = 0; i < LayerStacks; ++i)
    {
        if (!Detail::read_parameters(stream, network[i]))
            return false;
    }
    return stream && stream.peek() == std::ios::traits_type::eof();
}

template<typename Arch, typename Transformer>
bool Network<Arch, Transformer>::write_parameters(std::ostream&      stream,
                                                  const std::string& netDescription) const {
    if (!write_header(stream, Network::hash, netDescription))
        return false;
    if (!Detail::write_parameters(stream, *featureTransformer))
        return false;
    for (std::size_t i = 0; i < LayerStacks; ++i)
    {
        if (!Detail::write_parameters(stream, network[i]))
            return false;
    }
    return bool(stream);
}

// Explicit template instantiation

template class Network<
  NetworkArchitecture<TransformedFeatureDimensionsBig, L2Big, L3Big>,
  FeatureTransformer<TransformedFeatureDimensionsBig, &StateInfo::accumulatorBig>>;

template class Network<
  NetworkArchitecture<TransformedFeatureDimensionsSmall, L2Small, L3Small>,
  FeatureTransformer<TransformedFeatureDimensionsSmall, &StateInfo::accumulatorSmall>>;

}  // namespace Stockfish::Eval::NNUE

/*
  Stockfish, a UCI chess playing engine derived from Glaurung 2.1
  Copyright (C) 2004-2024 The Stockfish developers (see AUTHORS file)

  Stockfish is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  Stockfish is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// Code for calculating NNUE evaluation function

namespace Stockfish::Eval::NNUE {

void hint_common_parent_position(const Position&    pos,
                                 const Networks&    networks,
                                 AccumulatorCaches& caches) {
    if (Eval::use_smallnet(pos))
        networks.small.hint_common_access(pos, &caches.small);
    else
        networks.big.hint_common_access(pos, &caches.big);
}

namespace {
// Converts a Value into (centi)pawns and writes it in a buffer.
// The buffer must have capacity for at least 5 chars.
void format_cp_compact(Value v, char* buffer, const Position& pos) {

    buffer[0] = (v < 0 ? '-' : v > 0 ? '+' : ' ');

    int cp = std::abs(UCIEngine::to_cp(v, pos));
    if (cp >= 10000)
    {
        buffer[1] = '0' + cp / 10000;
        cp %= 10000;
        buffer[2] = '0' + cp / 1000;
        cp %= 1000;
        buffer[3] = '0' + cp / 100;
        buffer[4] = ' ';
    }
    else if (cp >= 1000)
    {
        buffer[1] = '0' + cp / 1000;
        cp %= 1000;
        buffer[2] = '0' + cp / 100;
        cp %= 100;
        buffer[3] = '.';
        buffer[4] = '0' + cp / 10;
    }
    else
    {
        buffer[1] = '0' + cp / 100;
        cp %= 100;
        buffer[2] = '.';
        buffer[3] = '0' + cp / 10;
        cp %= 10;
        buffer[4] = '0' + cp / 1;
    }
}

// Converts a Value into pawns, always keeping two decimals
void format_cp_aligned_dot(Value v, std::stringstream& stream, const Position& pos) {

    const double pawns = std::abs(0.01 * UCIEngine::to_cp(v, pos));

    stream << (v < 0   ? '-'
               : v > 0 ? '+'
                       : ' ')
           << std::setiosflags(std::ios::fixed) << std::setw(6) << std::setprecision(2) << pawns;
}
}

// Returns a string with the value of each piece on a board,
// and a table for (PSQT, Layers) values bucket by bucket.
std::string
trace(Position& pos, const Eval::NNUE::Networks& networks, Eval::NNUE::AccumulatorCaches& caches) {

    std::stringstream ss;

    char board[3 * 8 + 1][8 * 8 + 2];
    std::memset(board, ' ', sizeof(board));
    for (int row = 0; row < 3 * 8 + 1; ++row)
        board[row][8 * 8 + 1] = '\0';

    // A lambda to output one box of the board
    auto writeSquare = [&board, &pos](File file, Rank rank, Piece pc, Value value) {
        const int x = int(file) * 8;
        const int y = (7 - int(rank)) * 3;
        for (int i = 1; i < 8; ++i)
            board[y][x + i] = board[y + 3][x + i] = '-';
        for (int i = 1; i < 3; ++i)
            board[y + i][x] = board[y + i][x + 8] = '|';
        board[y][x] = board[y][x + 8] = board[y + 3][x + 8] = board[y + 3][x] = '+';
        if (pc != NO_PIECE)
            board[y + 1][x + 4] = PieceToChar[pc];
        if (value != VALUE_NONE)
            format_cp_compact(value, &board[y + 2][x + 2], pos);
    };

    // We estimate the value of each piece by doing a differential evaluation from
    // the current base eval, simulating the removal of the piece from its square.
    auto [psqt, positional] = networks.big.evaluate(pos, &caches.big);
    Value base              = psqt + positional;
    base                    = pos.side_to_move() == WHITE ? base : -base;

    for (File f = FILE_A; f <= FILE_H; ++f)
        for (Rank r = RANK_1; r <= RANK_8; ++r)
        {
            Square sq = make_square(f, r);
            Piece  pc = pos.piece_on(sq);
            Value  v  = VALUE_NONE;

            if (pc != NO_PIECE && type_of(pc) != KING)
            {
                auto st = pos.state();

                pos.remove_piece(sq);
                st->accumulatorBig.computed[WHITE] = st->accumulatorBig.computed[BLACK] = false;

                std::tie(psqt, positional) = networks.big.evaluate(pos, &caches.big);
                Value eval                 = psqt + positional;
                eval                       = pos.side_to_move() == WHITE ? eval : -eval;
                v                          = base - eval;

                pos.put_piece(pc, sq);
                st->accumulatorBig.computed[WHITE] = st->accumulatorBig.computed[BLACK] = false;
            }

            writeSquare(f, r, pc, v);
        }

    ss << " NNUE derived piece values:\n";
    for (int row = 0; row < 3 * 8 + 1; ++row)
        ss << board[row] << '\n';
    ss << '\n';

    auto t = networks.big.trace_evaluate(pos, &caches.big);

    ss << " NNUE network contributions "
       << (pos.side_to_move() == WHITE ? "(White to move)" : "(Black to move)") << std::endl
       << "+------------+------------+------------+------------+\n"
       << "|   Bucket   |  Material  | Positional |   Total    |\n"
       << "|            |   (PSQT)   |  (Layers)  |            |\n"
       << "+------------+------------+------------+------------+\n";

    for (std::size_t bucket = 0; bucket < LayerStacks; ++bucket)
    {
        ss << "|  " << bucket << "        "  //
           << " |  ";
        format_cp_aligned_dot(t.psqt[bucket], ss, pos);
        ss << "  "  //
           << " |  ";
        format_cp_aligned_dot(t.positional[bucket], ss, pos);
        ss << "  "  //
           << " |  ";
        format_cp_aligned_dot(t.psqt[bucket] + t.positional[bucket], ss, pos);
        ss << "  "  //
           << " |";
        if (bucket == t.correctBucket)
            ss << " <-- this bucket is used";
        ss << '\n';
    }

    ss << "+------------+------------+------------+------------+\n";

    return ss.str();
}
```

```
}  // namespace Stockfish::Eval::NNUE

// use the stockfish namespace //
using namespace Stockfish;

// the stockfish engine class //
class StockfishEngine
{
   // the command line stream //
   std::istringstream CommandLineStream;

   // the output line stream //
   std::istringstream OutputLineStream;

   // the intercepted output stream //
   std::stringstream CoutStream;

   // the output line string //
   std::string OutputLineString;

   // the best move found string //
   std::string BestMoveFound;

   // the information string //
   std::string InfoString;

   // the command token string //
   std::string CommandToken;

   // the output token string //
   std::string OutputToken;

   // the uci engine //
   UCIEngine *UCI;

   // the argument count //
   int argc = 1;

   // the arguments //
   char *argv[];

   // public functions //
   public:

   // initialize the stockfish engine //
   void StockfishInitialize()
   {
      // write out the engine information //
      std::cout << engine_info() << std::endl;

      // redirect the cout buffer //
      std::cout.rdbuf(CoutStream.rdbuf());

      // initialize the bitboards //
      Bitboards::init();

      // initialize the position //
      Position::init();

      // initialize the uci engine //
      UCI = new UCIEngine(argc, argv);

      // initialize the tune //
      Tune::init(UCI->engine_options());

      // start a new game //
      StockfishWrite("ucinewgame\n");
   }

   // write a command to the stockfish engine //
   void StockfishWrite(const char *Command)
   {
      // clear the command line stream //
      CommandLineStream.clear();

      // read the engine command into the command line stream //
      CommandLineStream.str(Command);

      // clear the command line token //
      CommandToken.clear();

      // skip over whitespace and extract a token from command line stream //
      CommandLineStream >> std::skipws >> CommandToken;

      // if need to stop //
      if (CommandToken == "stop")
      {
         // tell the engine to stop //
         UCI->engine.stop();
      }
      // if is for set option //
      else if (CommandToken == "setoption")
      {
         // set the specified options from the command line stream //
         UCI->setoption(CommandLineStream);
      }
      // if is for go //
      else if (CommandToken == "go")
      {
         // set that stockfish engine not done //
         StockfishIsDone = false;

         // reset the output stream //
         CoutStream.str(std::string());

         // clear the output stream //
         CoutStream.clear();

         // start searching from the current position //
         UCI->go(CommandLineStream);
      }
      // if is for the position //
      else if (CommandToken == "position")
      {
         // set the specified position //
         UCI->position(CommandLineStream);
      }
      // if is for starting a new game //
      else if (CommandToken == "ucinewgame")
      {
         // start a new game //
         UCI->engine.search_clear();
      }
   }

   // get the stockfish info string //
   string StockfishGetInfoString() {return InfoString;}

   // get the stockfish engine name //
   string StockfishGetEngineName() {return engine_info();}

   // find the best stockfish engine move //
   string StockfishFindBestMove()
   {
      // clear the best move found //
      BestMoveFound.clear();

      // clear the information string //
      InfoString.clear();

      // while stockfish is not done finding a best move //
      while (!StockfishIsDone)
      {
         // sleep for 100 milliseconds //
         std::this_thread::sleep_for(std::chrono::milliseconds(100));
      }

      // sleep for 100 milliseconds //
      std::this_thread::sleep_for(std::chrono::milliseconds(100));

      // while haven't found a best move yet //
      while (BestMoveFound.empty())
      {
         // clear the output stream //
         CoutStream.clear();

         // clear the output line string //
         OutputLineString.clear();

         // get a new line from the output stream //
         std::getline(CoutStream, OutputLineString);

         // if the output line string is not empty //
         if (!OutputLineString.empty())
         {
            // clear the output line stream //
            OutputLineStream.clear();

            // set the output line stream to the output line string //
            OutputLineStream.str(OutputLineString);

            // clear the output token //
            OutputToken.clear();

            // extract the first element from the output line stream //
            OutputLineStream >> std::skipws >> OutputToken;

            // if is line with the best move //
            if (OutputToken == "bestmove")
            {
               // save the best move found //
               OutputLineStream >> std::skipws >> BestMoveFound;
            }
            // if is line which contains information with principal variation //
            else if (OutputToken == "info")
            {
               // save information string that contains principal variation //
               InfoString = OutputLineString;
            }
         }
      }

      // return the best move found //
      return BestMoveFound;
   }
};

// wrapper to create a new stockfish engine //
void *StockfishEngineNew() {return new StockfishEngine();}

// wrapper to initialize the stockfish engine //
void StockfishInitialize(void *Stockfish)
{
   // initialize the stockfish engine //
   ((StockfishEngine *) Stockfish)->StockfishInitialize();
}

// wrapper to write a command to the stockfish engine //
void StockfishWrite(void *Stockfish, const char *Command)
{
   // write a command to the stockfish engine //
   ((StockfishEngine *) Stockfish)->StockfishWrite(Command);
}

// wrapper to find the stockfish info string //
char *StockfishGetInfoString(void *Stockfish)
{
   // get the info string //
   std::string InfoString = ((StockfishEngine *) Stockfish)
                            ->StockfishGetInfoString();

   // return the info string //
   return std::strcpy((char *) std::malloc(InfoString.length() + 1), InfoString
          .c_str());
}

// wrapper to find the stockfish engine name string //
char *StockfishGetEngineName(void *Stockfish)
{
   // get the engine name string //
   std::string EngineName = ((StockfishEngine *) Stockfish)
                            ->StockfishGetEngineName();

   // return the engine name string //
   return std::strcpy((char *) std::malloc(EngineName.length() + 1), EngineName
          .c_str());
}

// wrapper to find the best stockfish engine move //
char *StockfishFindBestMove(void *Stockfish)
{
   // get the best move string //
   std::string BestMove = ((StockfishEngine *) Stockfish)
                          ->StockfishFindBestMove();

   // return the best move string //
   return std::strcpy((char *) std::malloc(BestMove.length() + 1), BestMove
          .c_str());
}
```