Implementing Lockless Look Up Tables in a NegaMax Search

Outline

- Key point: Lockless lookup table – Implementation
  - We need a context!
- Some basics
- Search algorithm: NegaMax
- Lockless lookup table

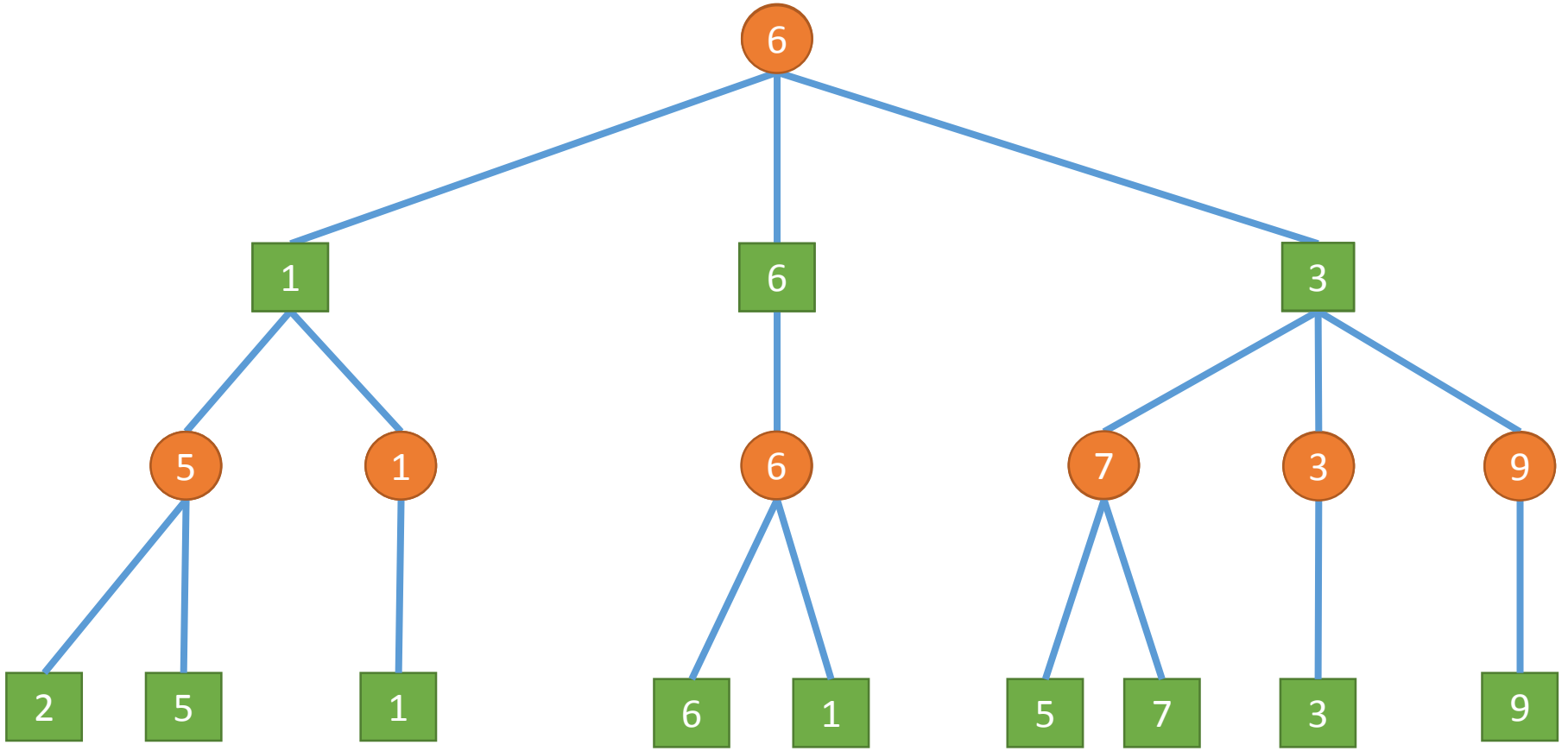
More >= Less

Elements

- Board
  - GenerateMoves()
  - MakeMove()
  - UndoMove()
- **Search()**
- Evaluate()

NegaMax == Compact form of MiniMax

MiniMax



MiniMax Pseudo Code

```
int maxi( int depth ) {
        if ( depth == 0 ) return evaluate();
        int max = -oo;
        for ( all moves) {
                score = mini( depth – 1 );
                if( score > max )
                        max = score;
        }
        return max;
}
```

```
int mini( int depth ) {
        if ( depth == 0 ) return -evaluate();
        int min = +oo;
        for ( all moves) {
                score = maxi( depth – 1 );
                if( score < min )
                        min = score;
        }
        return min;
}
```

NegaMax Pseudo Code

```
int negaMax( int depth ) {
      if ( depth == 0 ) return evaluate();
      int max = -oo;
      for ( all moves)  {
            score = -negaMax( depth – 1 );
            if( score > max )
                  max = score;
      }
      return max;
}
```

Using NegaMax

- Root
- Actual

Root Pseudo Code:

```
int root( int depth ) {
        if ( depth == 0 ) return evaluate();
        int max = -oo, score;
        for ( all moves)  {
                make move();
                score = -negaMax(depth – 1);
                if(score > max)
                {
                        max = score;
                }
        }
        return max;
}
```

Parallelised NegaMax

- Parallel **`for`** loop in the **`root`** negamax method
- Results of all iterations go in a separate array
- Look for maximum value in that array

➢One array slot for every iteration

Transposition Table

- MiniMax Tree has cycles in it

- Cache search results in a lookup table, called transposition table in chess

➢ **Key:** Board Position

➢ **Value:** Score, Depth, Best Move

Problem

How to use a single transposition table in a parallelised search?

Solutions

1. Sync locks
2. **Lockless transposition table**

Lockless Transposition Table

- Key: Zobrist Hash (64 bit word)
- Value: A 64-bit word

➢In C/C++ the 64-bit word will be **`unsigned long long`**

Lockless Transposition Table

- Dr Robert Hyatt's article: http://www.cis.uab.edu/hyatt/hashing.html
- Our solution is for x86_64 architectures

Zobrist Hash

- 64-bit hash of a board position in chess

board position == positions of every piece + side to move + castling rights + en passant

- Like any hash, it has little chance of clash

➢Hashes of two similar positions will be far apart

Exact Problem

- We have 64-bit key and a 64-bit value
- On x86_64 architecture, 64-bit reads and writes are atomic
- Therefore: when read, a key or a value will either be new or original, but not half-way in-between

- Old Key – Old Value
- New Key – New Value
- **Old Key – New Value**
- **New Key – Old Value**

How It Works

- XOR is a reversible operation:

```
X ^ Y ^ Y == X
```

- For example:

```
X = 1011; Y = 0101;
X ^ Y = 1011 ^ 0101 = 1110
X ^ Y ^ Y = 1110 ^ 0101 = 1011 = X
```

How It Works

- Modify how we store keys:

```
Key = Zobrist Hash ^ 64-bit Value
```

- Retrieve as follows:

```
if(Key ^ 64-bit Value == Search Zobrist Hash)
        Value is fine; // so use it
else
      Value is corrupted; // discard it
```

Using Transposition Table

- Size of the table is limited
- One scheme to enter values into the table:
  - Zobrist hash modulo length of the table

➢Zobrist hash helps keep similar board positions in the table for longer

For Further Exploration...

- https://github.com/bytefire/Shutranj
  - C#, NegaMax
- A better engine called Stockfish: https://github.com/mcostalba/Stockfish
  - C++, a variant of Alpha-Beta