CHESS ENGINE

F. Ivone – Winter Seminar 2024

WHAT?

- A **chess engine** is a computer program that can analyze chess positions and provides a numeric evaluation of the chances of victory (score).

- In 1997, for the first time, a chess engine named Deep Blue defeated **Garry Kasparov, the world champion.**

- Currently the strongest chess engine is called Stockfish, it is open source (https://github.com/official-stockfish/Stockfish/) and outperforms any human player by a huge margin.

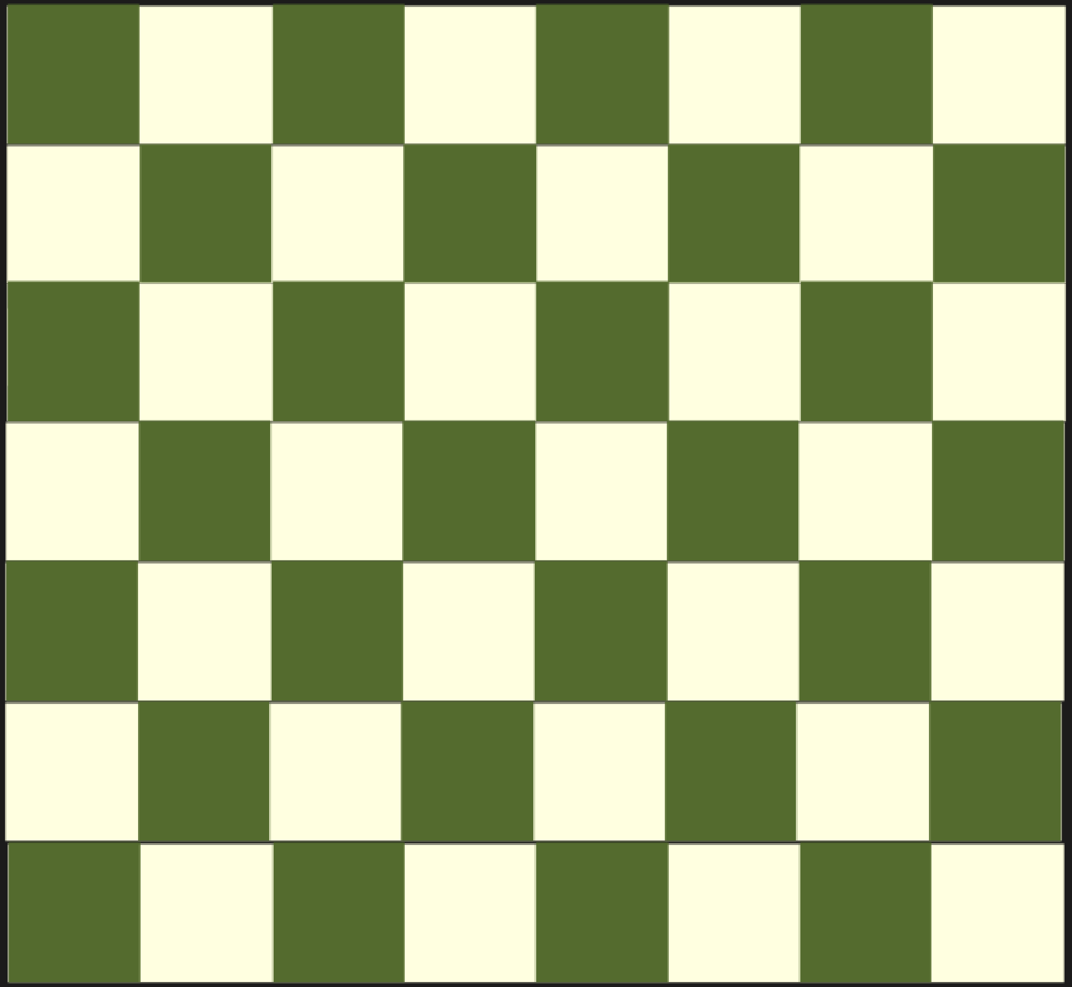
WHY?

Why not!

Clear Progress Evaluation
Engines, or engine versions, can compete to each other, leading a clear rating as changes are deployed.

Coding Challenge
I wanted to explore new techniques on C++ and new optimization algorithms.

CHESSBOARD REPRESENTATION



A **chessboard** is made of 64 squares. Each square is in a binary state: either **occupied** by a piece or **empty**.

How to represent it in the code?

CHESSBOARD REPRESENTATION

A **64 bit** number can be thought as 64 squares in a binary state: either **ON** or **OFF.**
This representation is called **bitboard**.

For example: a chessboard occupied on squares 3 and 55 can be mapped to

`00000000 01000000 00000000 00000000 00000000 00000000 00000000 00000100`

CHESSBOARD REPRESENTATION

A **64 bit** number can be thought as 64 squares in a binary state: either **ON** or **OFF.**
This representation is called **bitboard**.

For example: a chessboard occupied on squares 3 and 55 can be mapped to

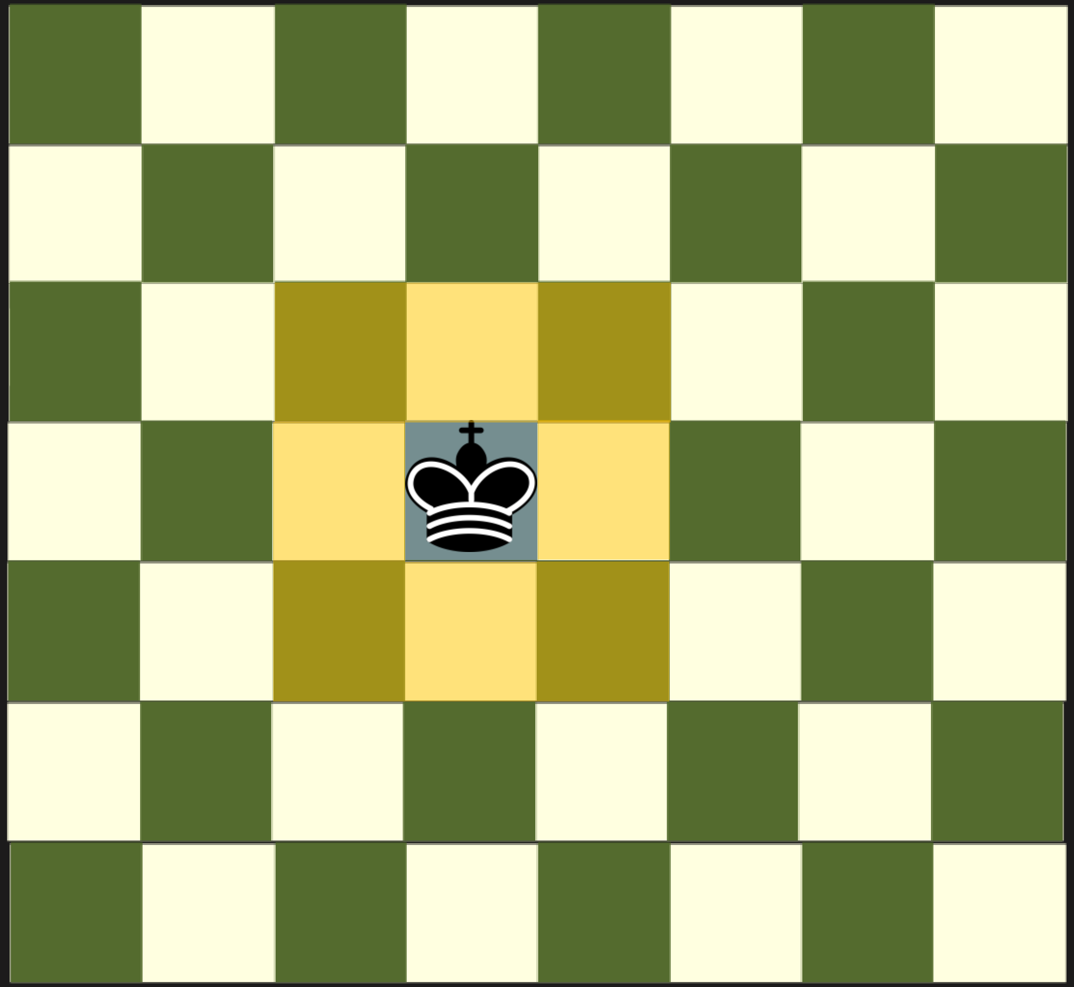
```
00000000 01000000 00000000 00000000
00000000 00000000 00000000 00000100
```

Simple bit operations (i.e. >>) will move all the piece on the board at once.

Modern processors are extremely fast at performing 64 bit instructions.

MOVE PIECES

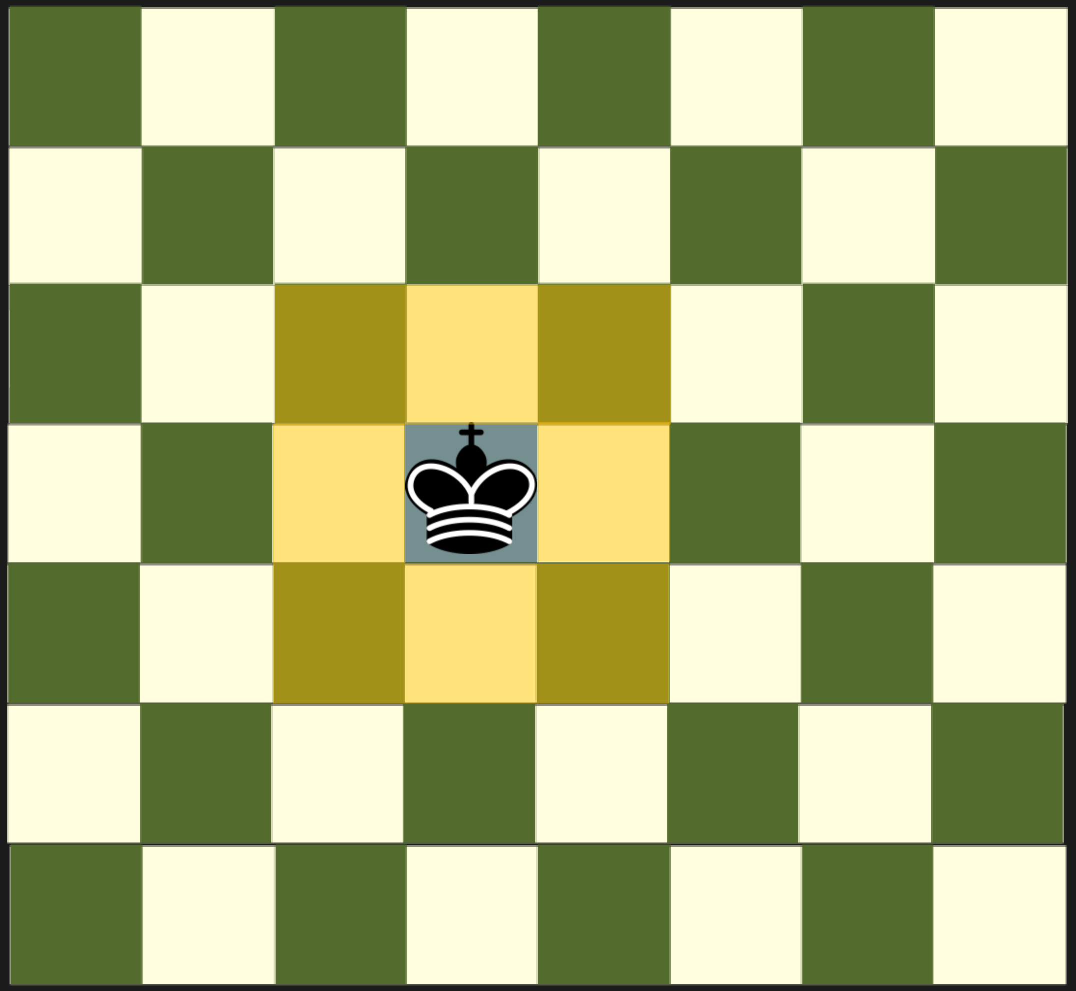
MOVE PIECES



There are 6 different piece types, each with specific move features.
Piece's movement can be **generated** with for loops. For example

```
for each piece:
     calculate possible moves
```

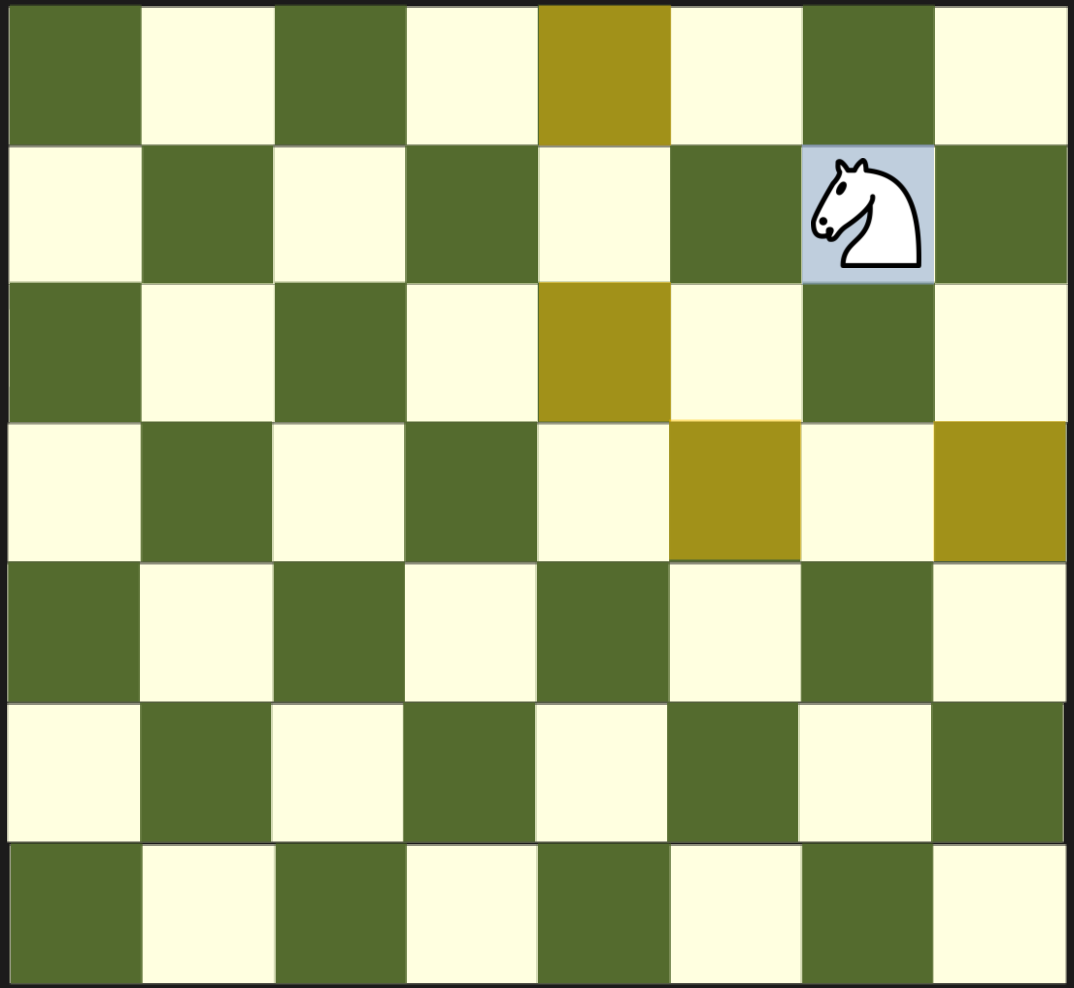
MOVE PIECES



There are 6 different piece types, each with specific move features.
Piece’s movement can be generated with for loops.

Faster to use **Look Up Tables (LUT)** containing pre-calculated moves for each piece type in any of the 64 square…

MOVE PIECES



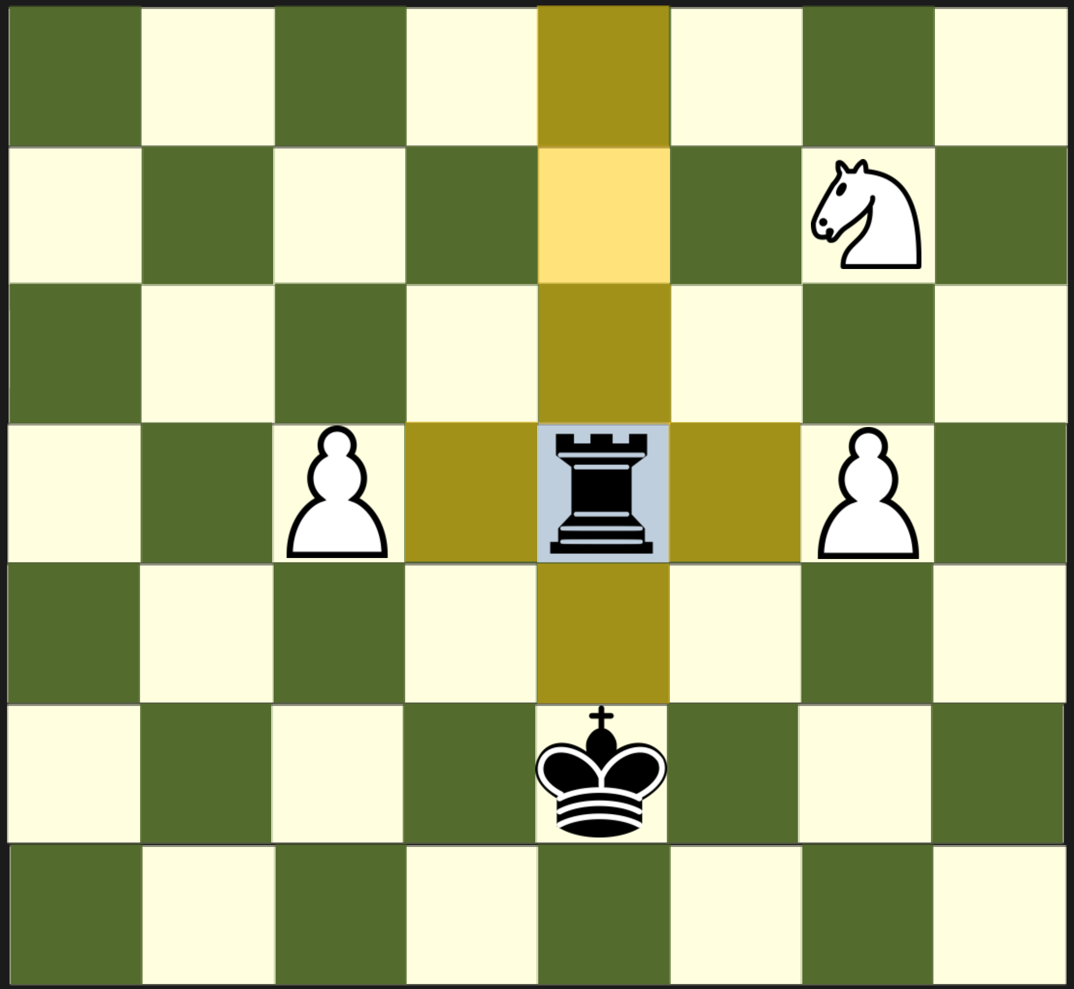
A look up table is a simple map

- from each of the 64 squares;
- to a **bitboard** representing a piece's moves expressed in the bitboard representation.

`[1,…,64] → [0xd2f,…,0x4a1f]`

Things get more complicated for "leaping" pieces: rook, bishop, queen.

MOVE PIECES

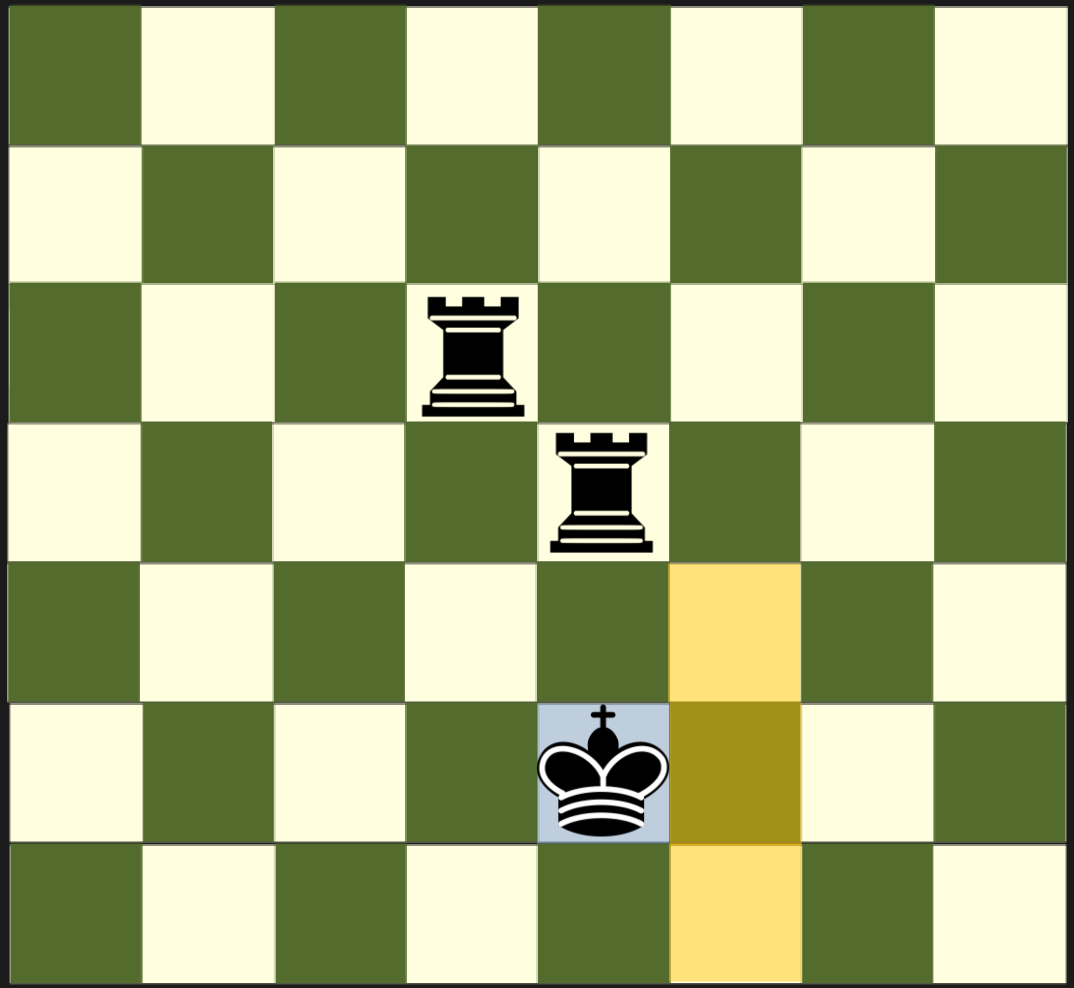


Leaping pieces can slide on the board until another piece is encountered aka a **blocker.**

It is still possible to use **LUT** to pre-calculate moves.

For **each square**, the configuration of all **relevant** blockers is hashed into an index [0,…,4096). The moves are then precalculated and stored in a
2D Look Up Table sized **64 x 4096.**

MOVE PIECES



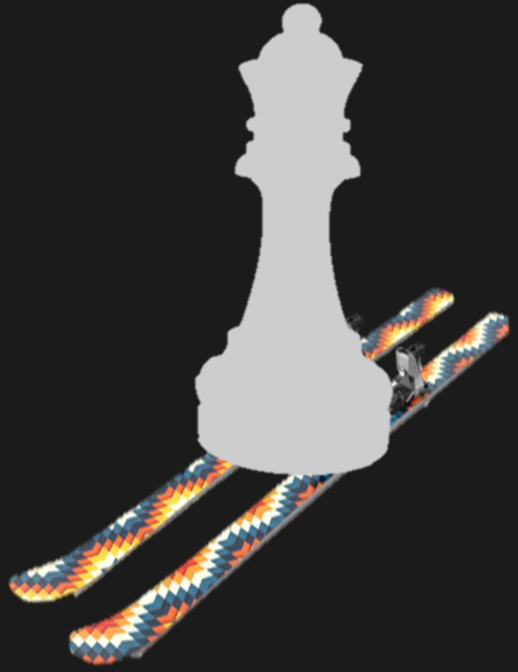
Not all **LUT** moves are legal moves.

Implement the chess rules to filter the LUT moves ➔ legal moves generator.

Picking a random move from a set of legal moves is not a great idea for a chess engine.

How to pick the best move?

MOVE SEARCH



MOVE SEARCH

Two basic ingredients for finding the best move

1. Evaluation (not in this talk)

Function to determine the score of a position.

The score is related to the chances of winning.

It is based on board occupancy, “chess good practices” and/or Neural Networks.

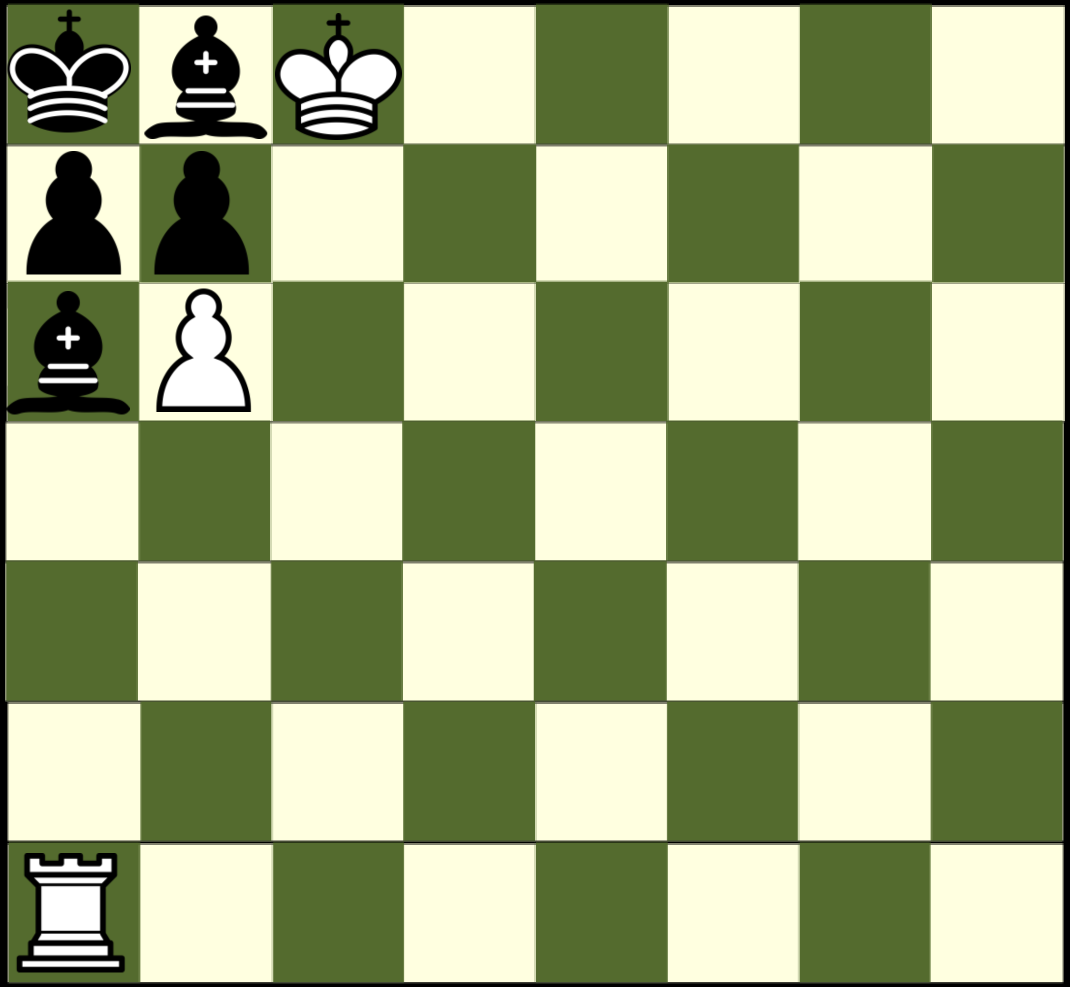
2. Maximizing algorithm

Recursive algorithm for choosing the optimal move for the current side playing.

Assumes optimal play from the opponent as well.

Scans the game tree up to a fixed depth.

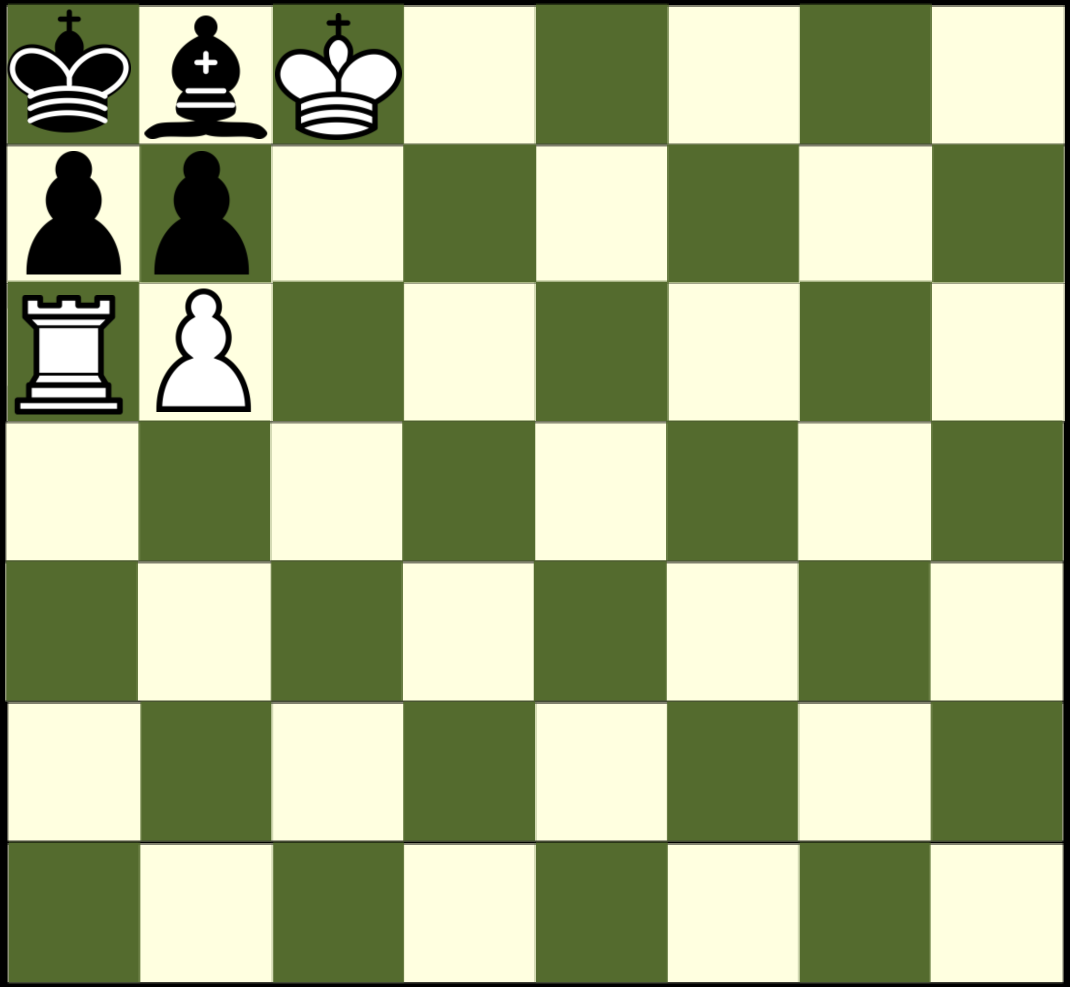
MOVE SEARCH



In this position **black** has more material. So the static evaluation would return "black has more chances to win".

However it is **white** turn to move and it is guaranteed a win in 3 moves.

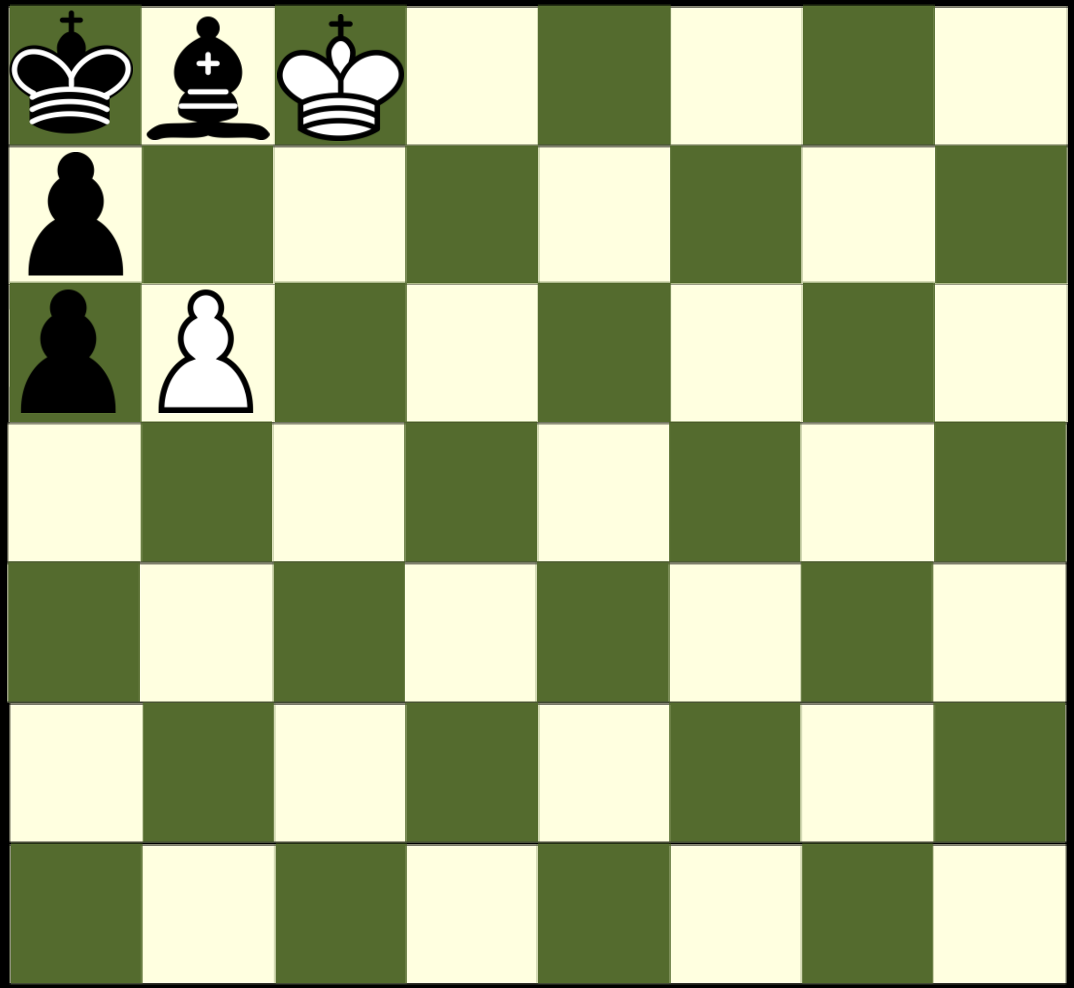
MOVE SEARCH



In this position **black** has more material. So the static evaluation would return "black has more chances to win".

However it is **white** turn to move and it is guaranteed a win in 3 moves.

MOVE SEARCH

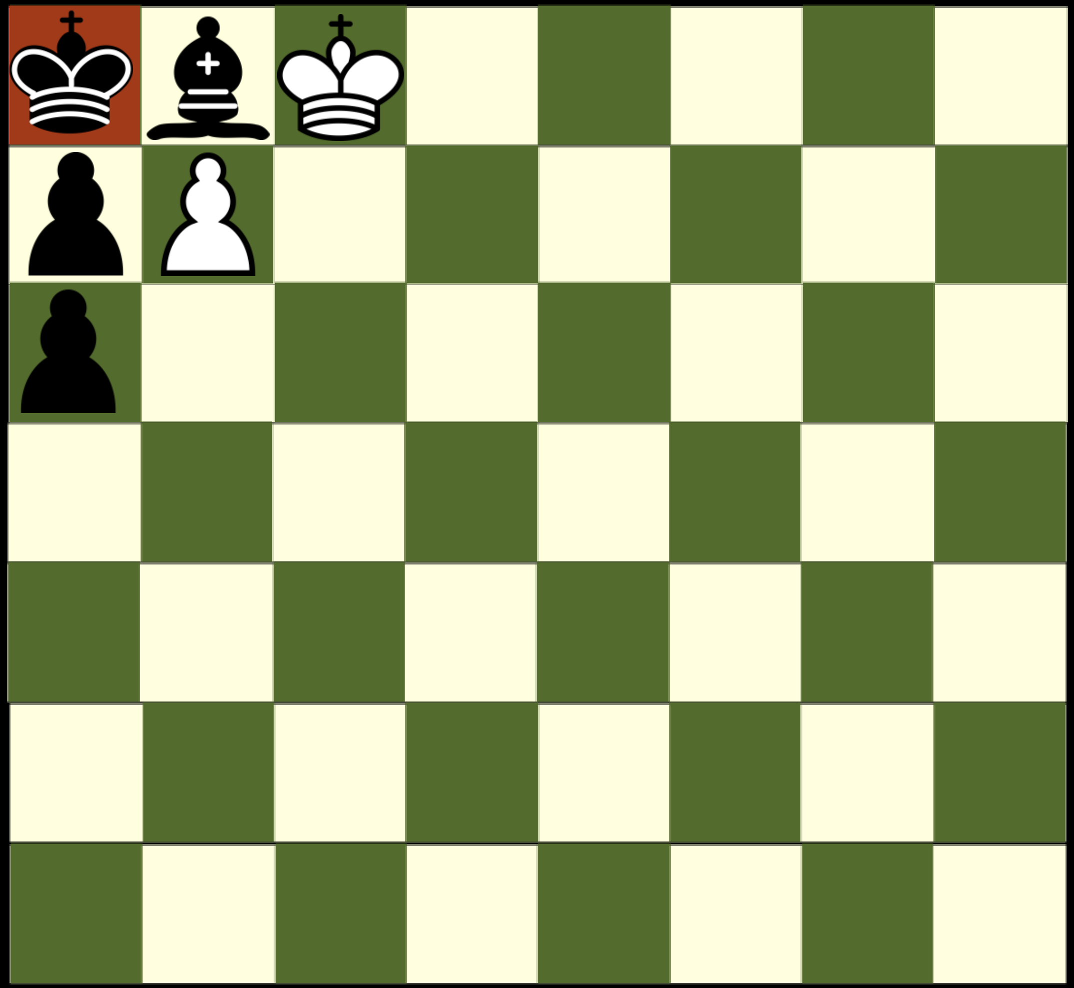


In this position **black** has more material. So the static evaluation would return “black has more chances to win”.

However it is **white** turn to move and it is guaranteed a win in 3 moves.

MOVE SEARCH

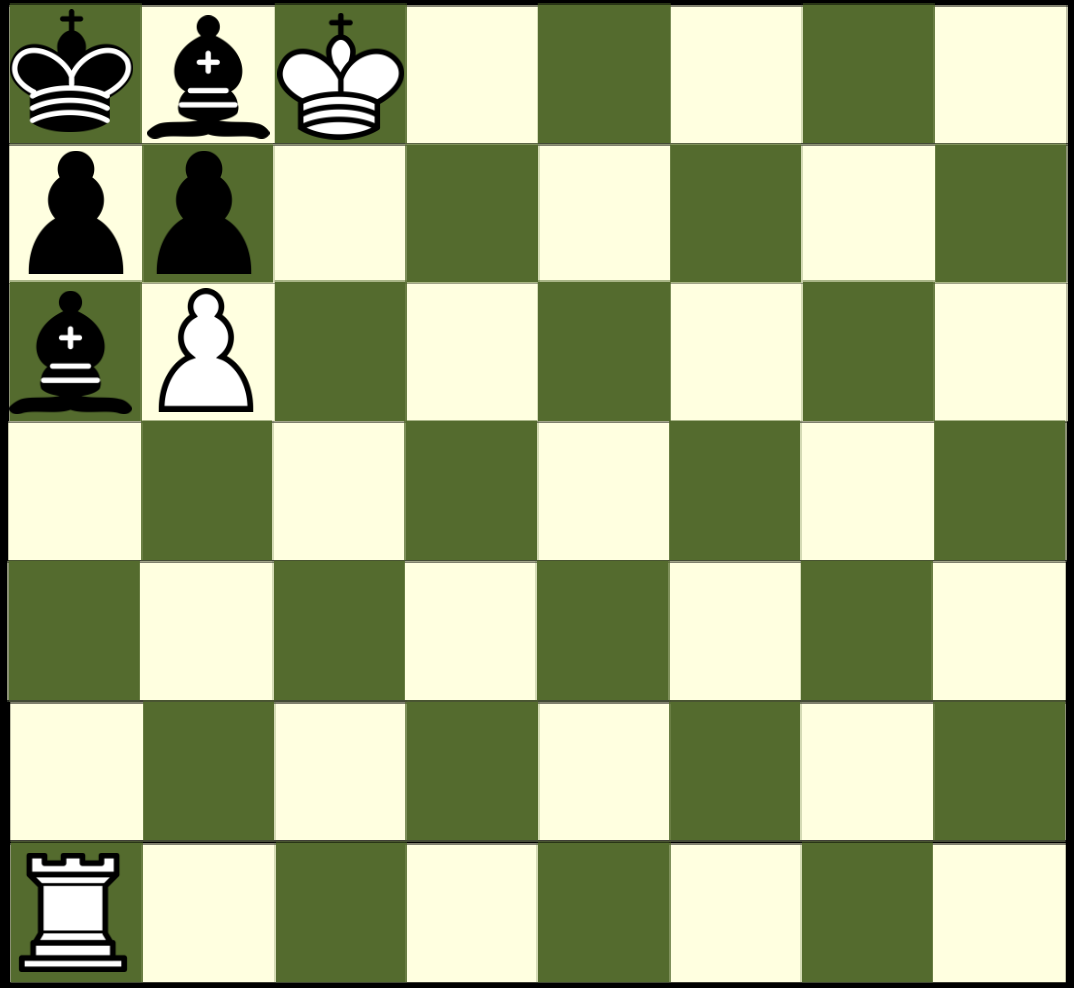
White wins



In this position **black** has more material. So the static evaluation would return “black has more chances to win”.

However it is **white** turn to move and it is guaranteed a win in 3 moves.

MOVE SEARCH

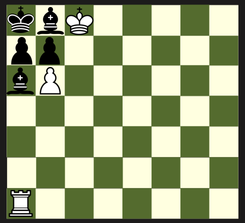


In this position **black** has more material.
So the static evaluation would return "black has more chances to win".

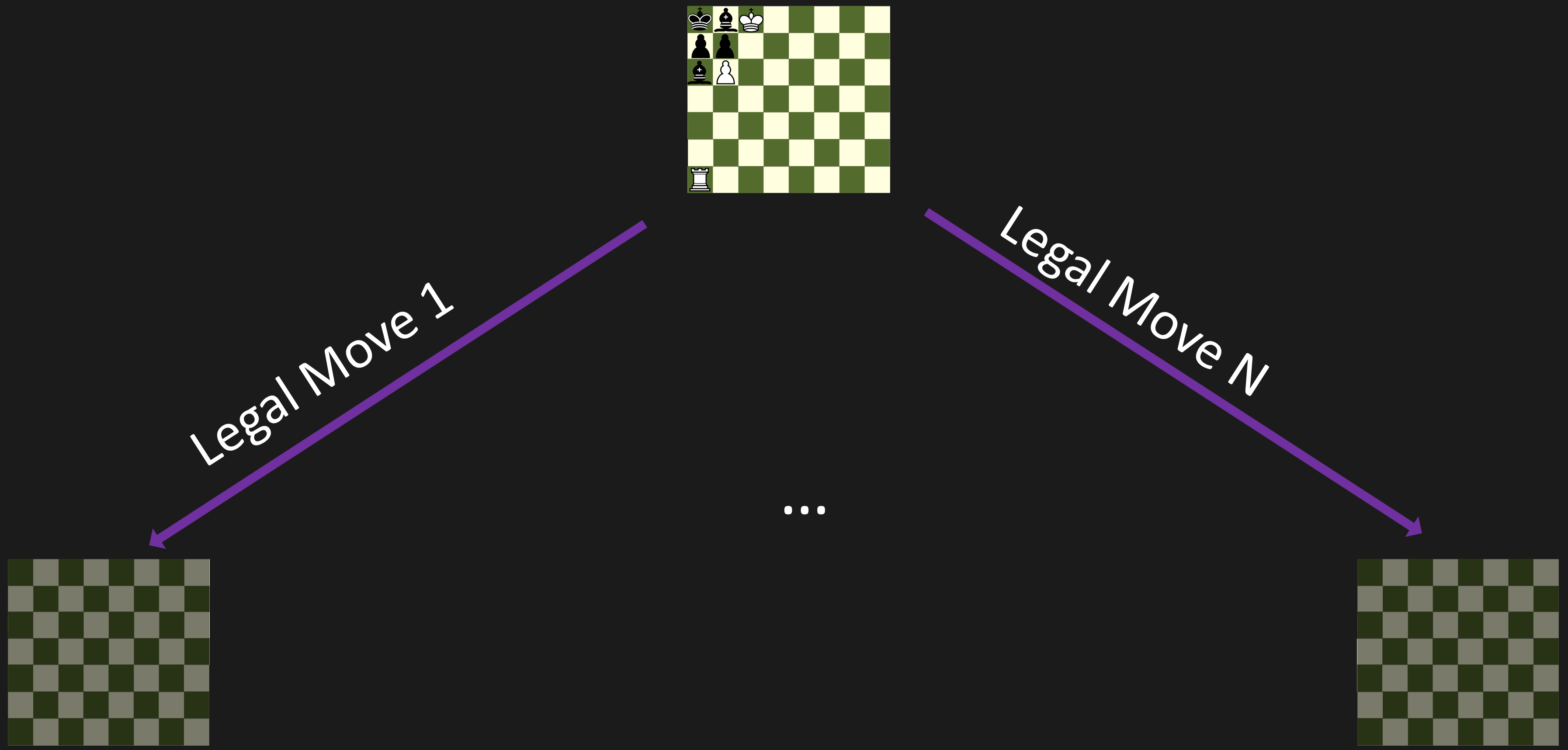
However it is **white** turn to move and it is guaranteed a win in 3 moves.
The static evaluation of the position is **NOT** accurate enough.

The **maximizing algorithm** looks into the future to provide a better position evaluation (dynamic).

THE MINIMAX ALGORITHM

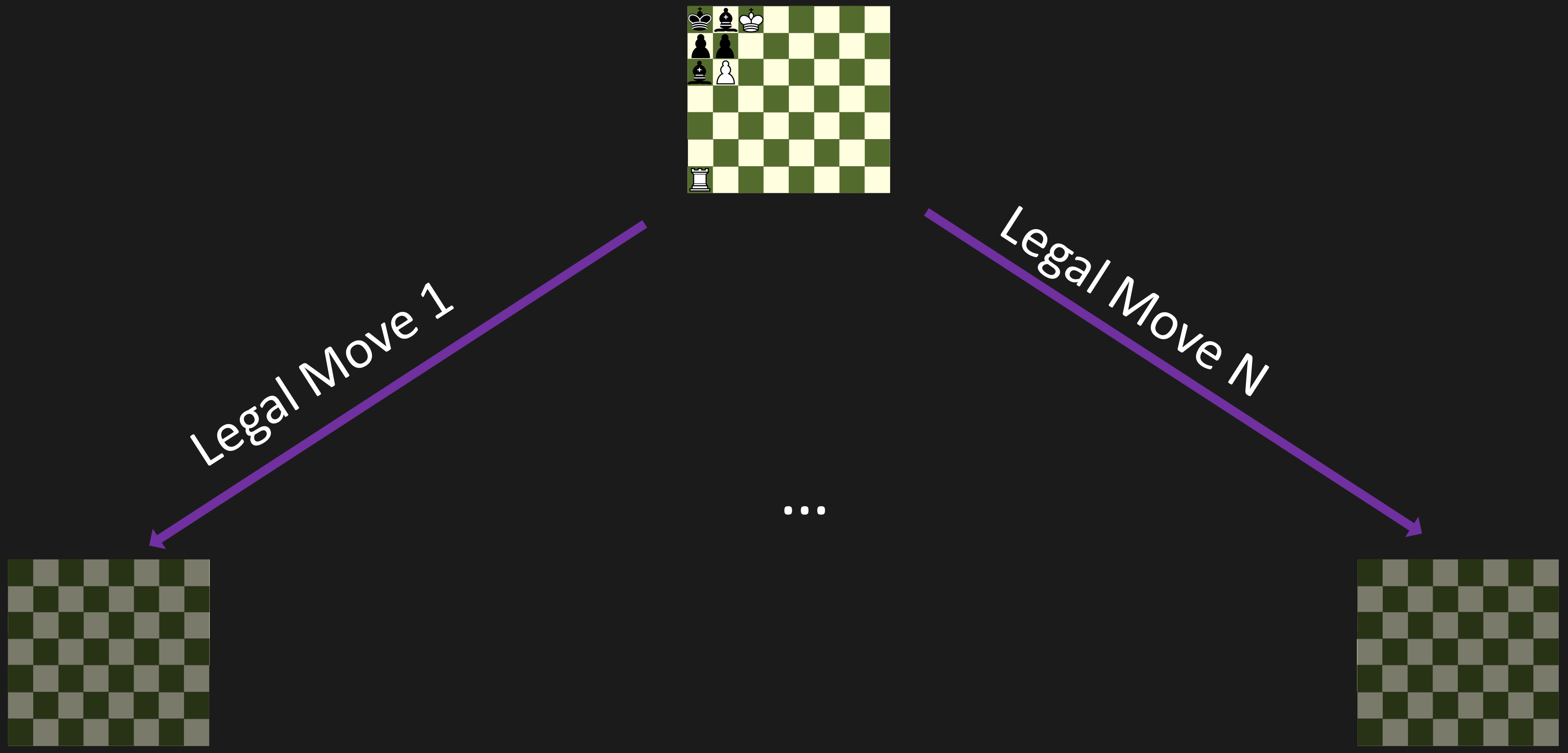


THE MINIMAX ALGORITHM



Depth 1
B to play

THE MINIMAX ALGORITHM

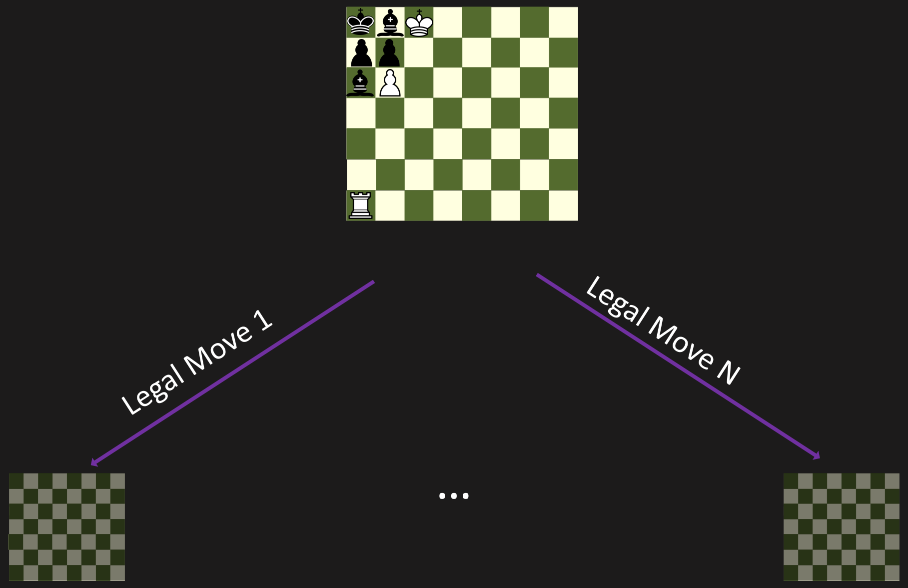


Depth 1
B to play

`current searched depth < 3`

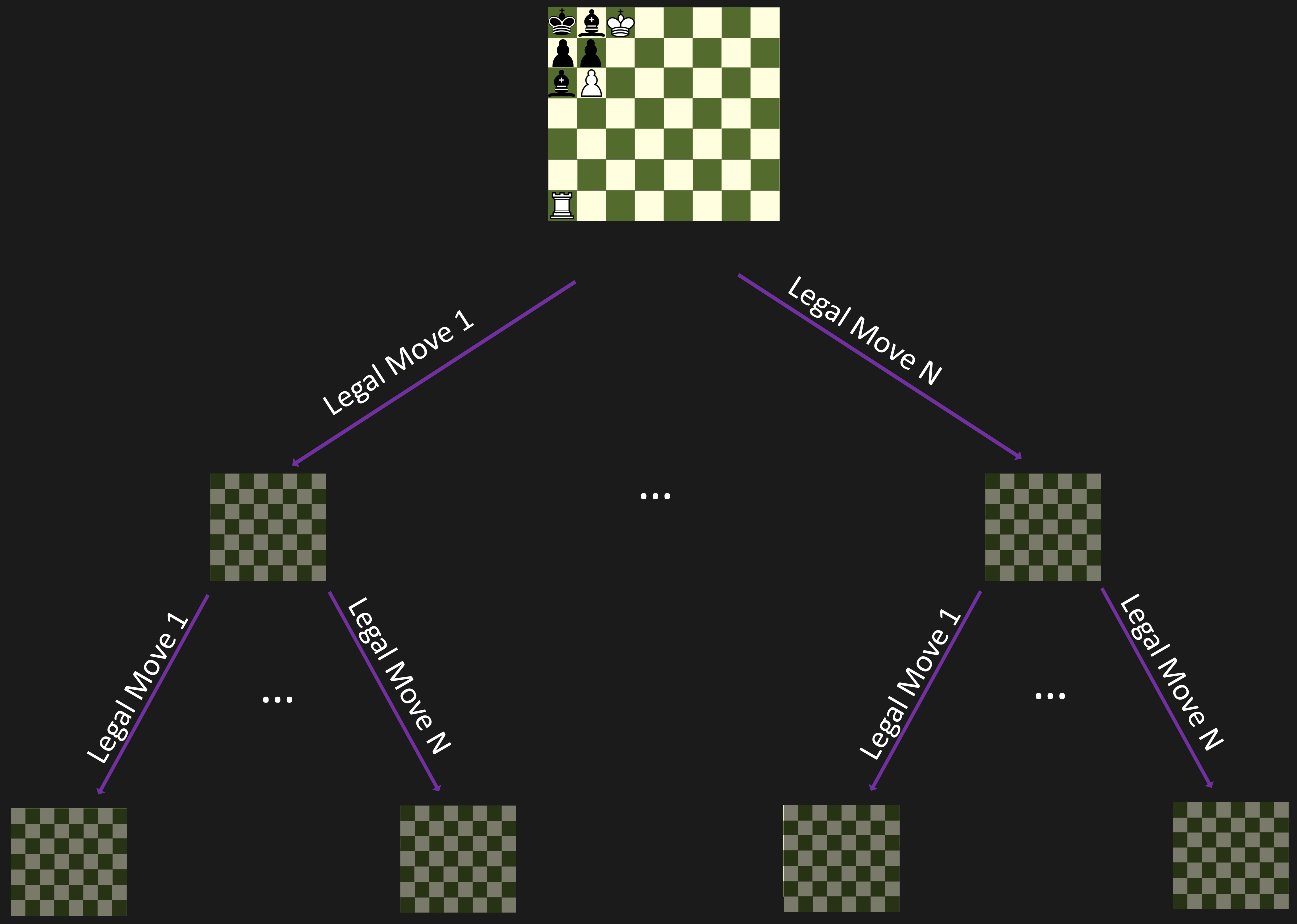
Therefore, minimax gets called again from all reached positions. Now it searches for the best **opponent** move.

THE MINIMAX ALGORITHM



Depth 1
B to play

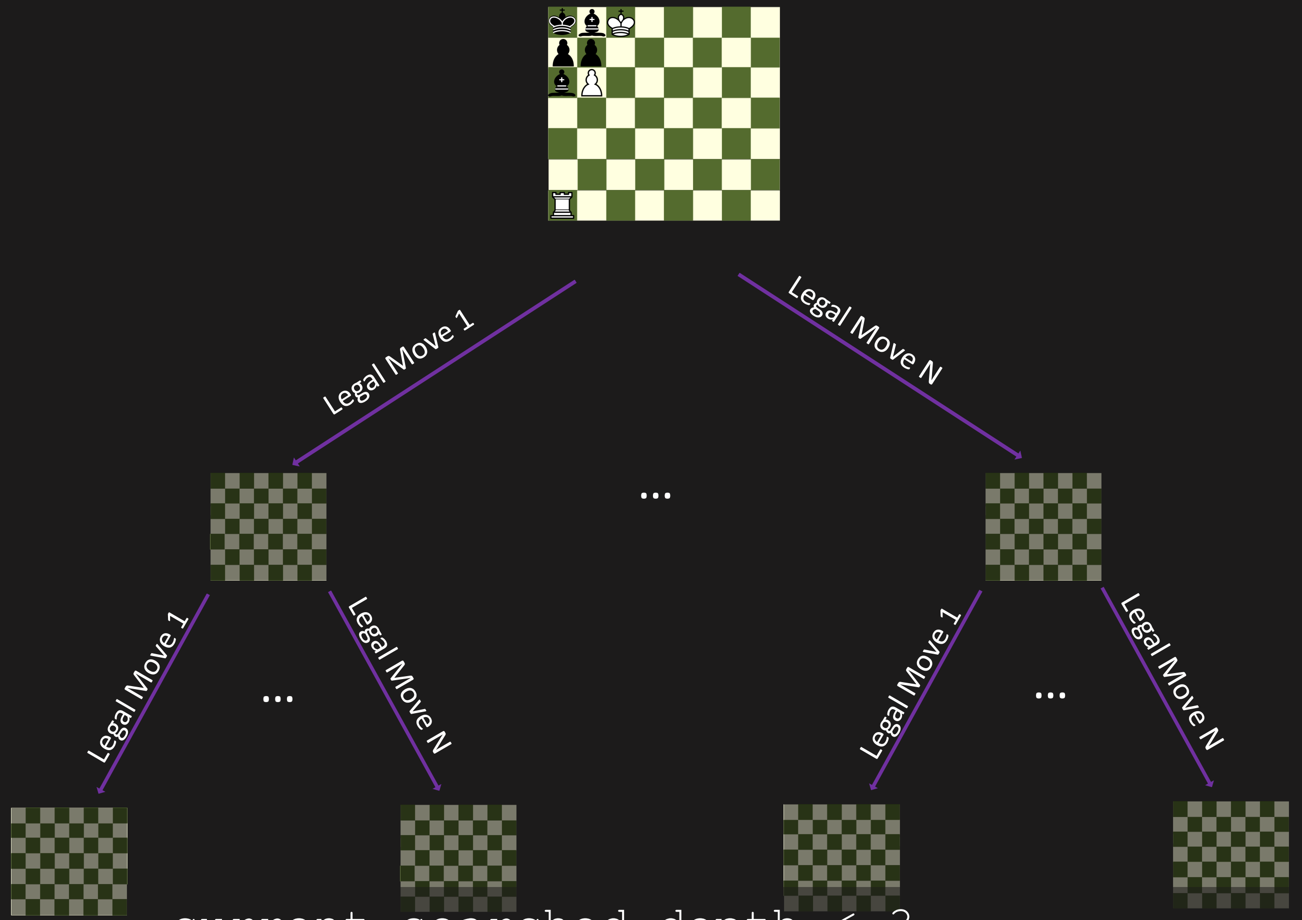
THE MINIMAX ALGORITHM



Depth 1
B to play

Depth 2
W to play

THE MINIMAX ALGORITHM



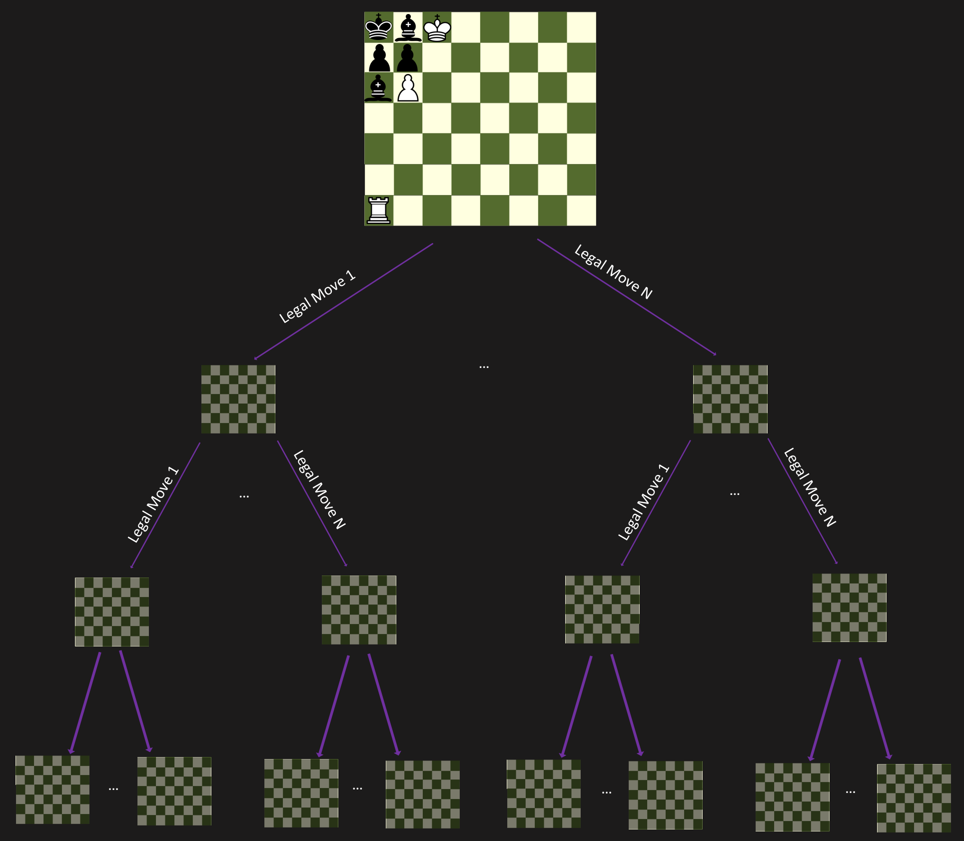
Depth 1
B to play

Depth 2
W to play

`current searched depth < 3`

Therefore, minimax gets called again from all reached positions. Now it searches for the best **opponent** move.

THE MINIMAX ALGORITHM



Depth 1
B to play

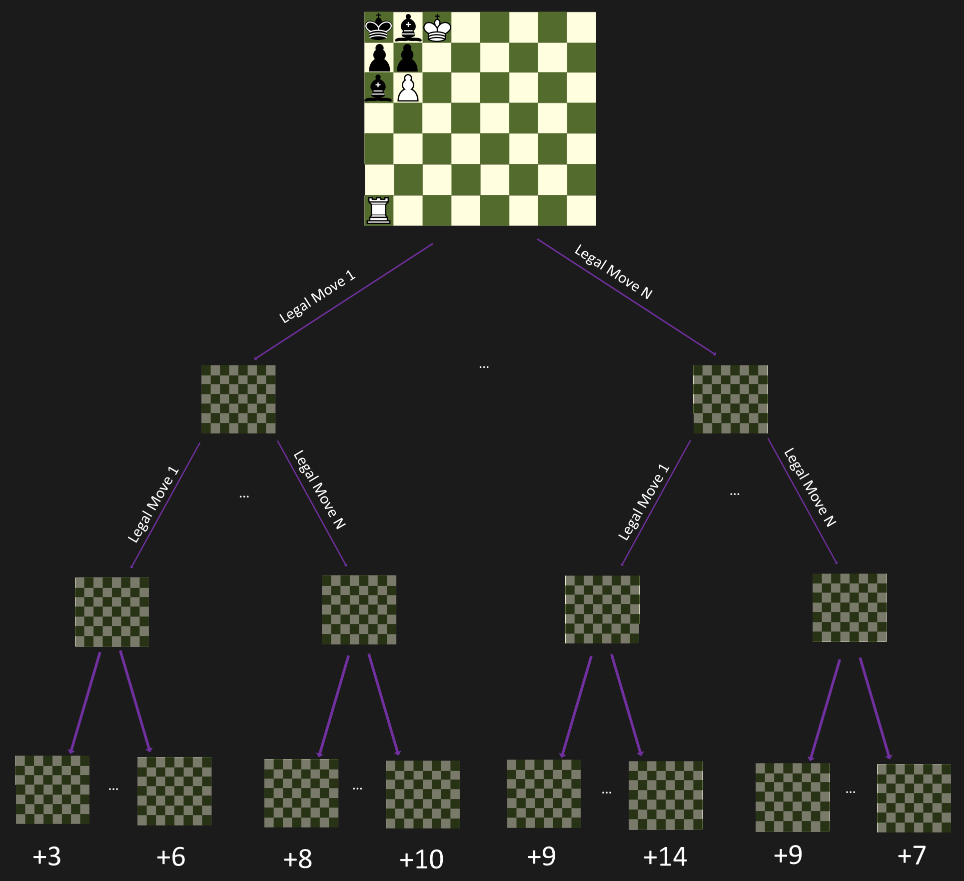
Depth 2
W to play

Depth 3
B to play

`current searched depth == 3`

These are now terminal nodes! The static evaluation is applied to return a score.

THE MINIMAX ALGORITHM

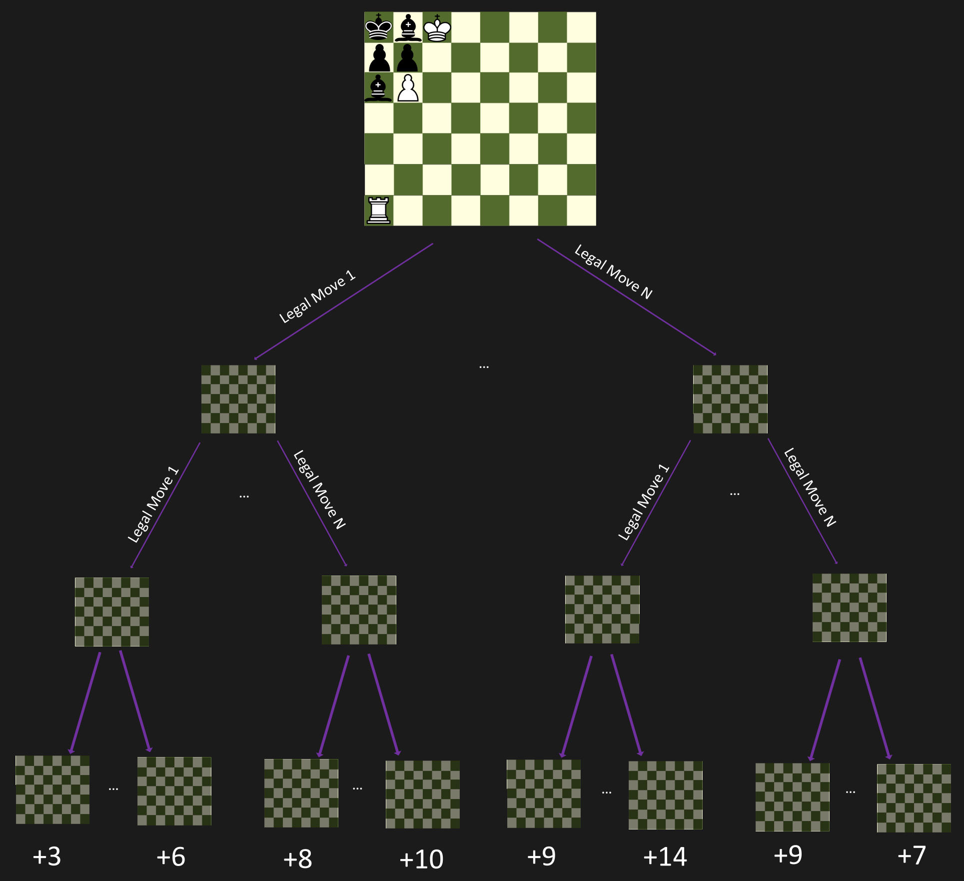


Depth 1
B to play

Depth 2
W to play

Depth 3
B to play

THE MINIMAX ALGORITHM



Depth 1
B to play

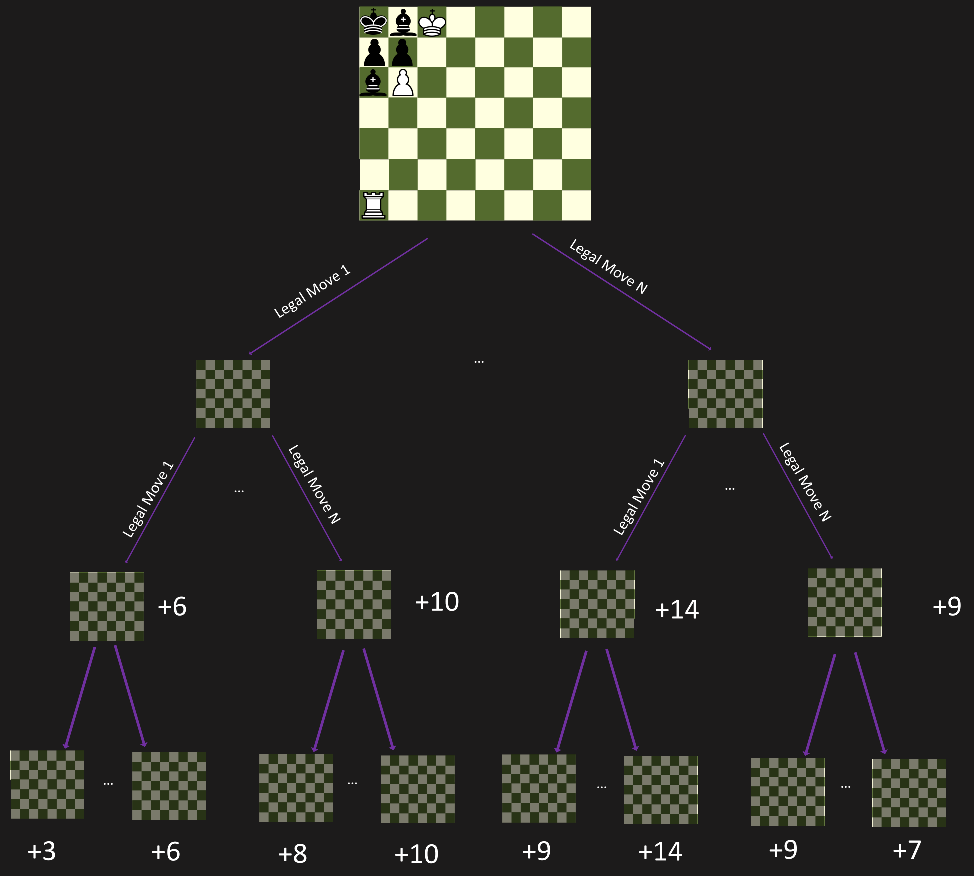
Depth 2
W to play

Depth 3
B to play

At depth 2, **white** has now several rated moves to choose from.
It will always pick the move that **maximizes** the score

THE MINIMAX ALGORITHM

At depth 2, **white** has now several rated moves to choose from.
It will always pick the move that **maximizes** the score



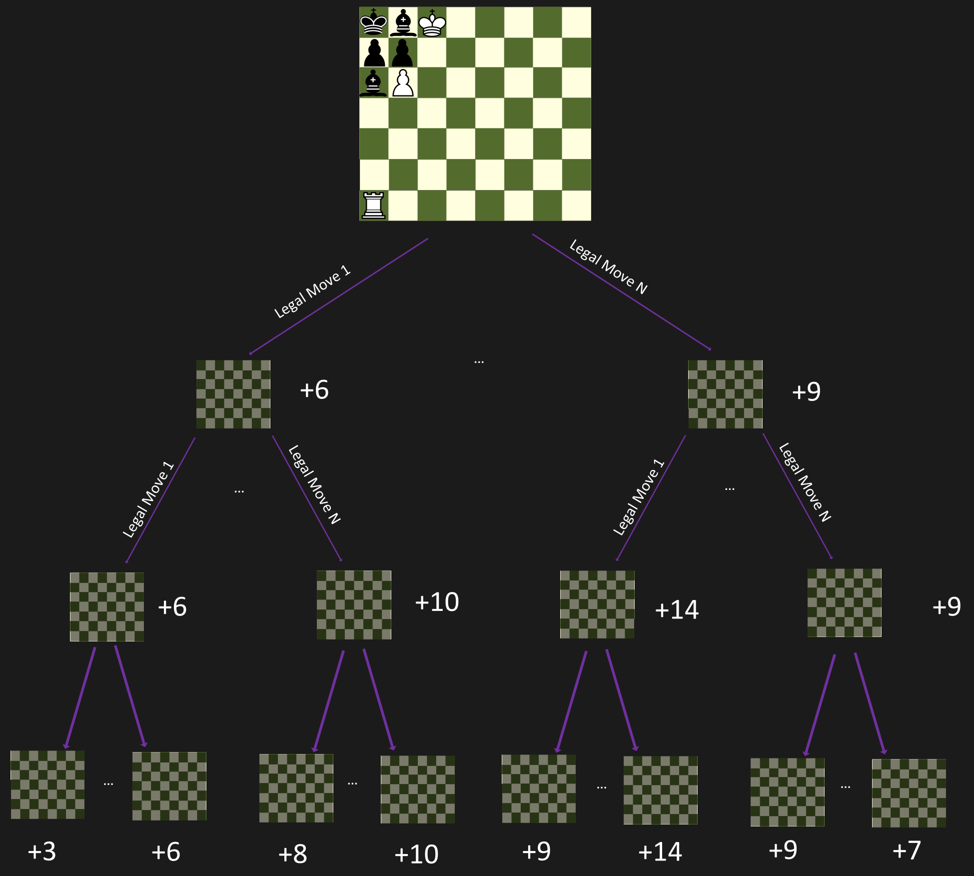
Depth 1
B to play

Depth 2
W to play

Depth 3
B to play

THE MINIMAX ALGORITHM

At depth 1, **black** has now several rated moves to choose from.
It will always pick the move that **minimizes** the score



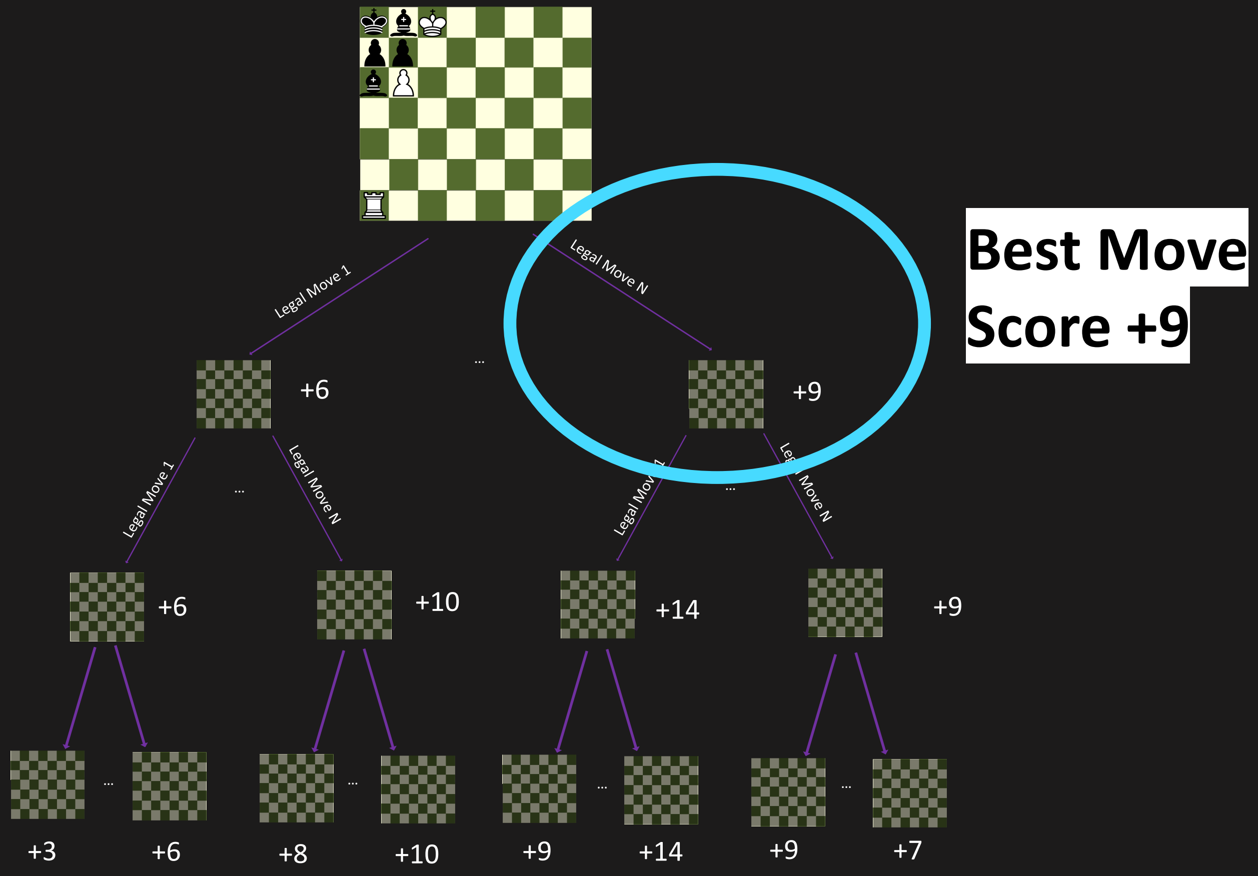
Depth 1
B to play

Depth 2
W to play

Depth 3
B to play

THE MINIMAX ALGORITHM

At depth 0, **white** has now several rated moves to choose from.
It will always pick the move that **maximizes** the score



Depth 1
B to play

Depth 2
W to play

Depth 3
B to play

THE MINIMAX ALGORITHM

- Recursive
- Static evaluation at terminal nodes
- Assumes best play by both players (max player and min player)

THE MINIMAX ALGORITHM

- Recursive
- Static evaluation at terminal nodes
- Assumes best play by both players
  (max player and min player)

Issues

The number of position (nodes) searched, grows exponentially. The computation time also does.

The search is depth limited.
A negative event might be unavoidable but delayable beyond the searched depth.

THE MINIMAX ALGORITHM

- Recursive
- Static evaluation at terminal nodes
- Assumes best play by both players (max player and min player)

Issues

The number of position (nodes) searched, grows exponentially. The computation time also does.

- Alpha Beta pruning
- Transposition table (touched on in this talk)

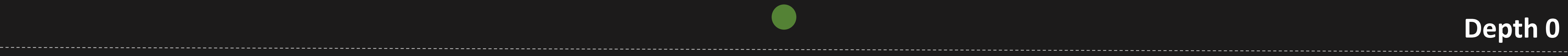
The search is depth limited.
A negative event might be unavoidable but delayable beyond the searched depth.

- Quiescence search (not in this talk)

ALPHA BETA PRUNING

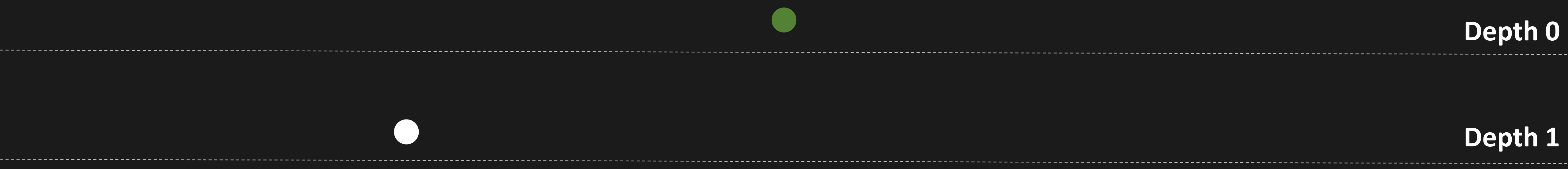
- Let’s say that during our search for the best move, we find a very good one.
- Then, we don’t need to search all remaining moves, but only those who are NOT worse than the current move found.
- As a result, we can ignore large portions of the game tree and speed up our search.

ALPHA BETA PRUNING



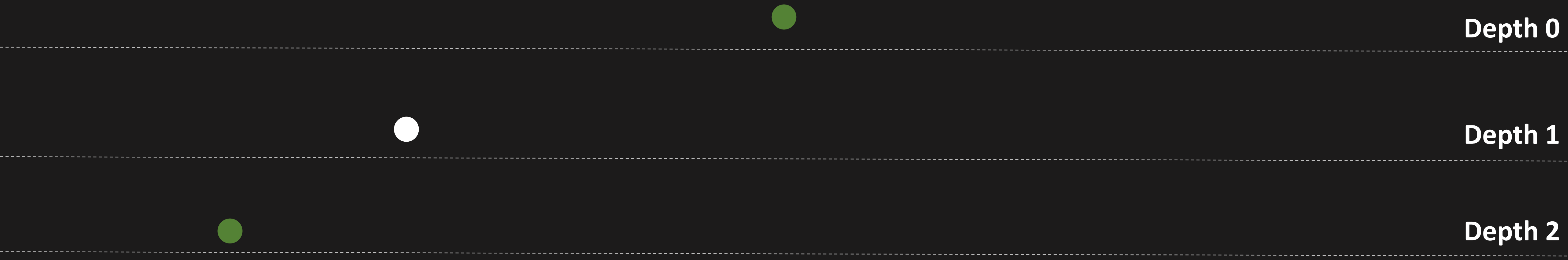
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



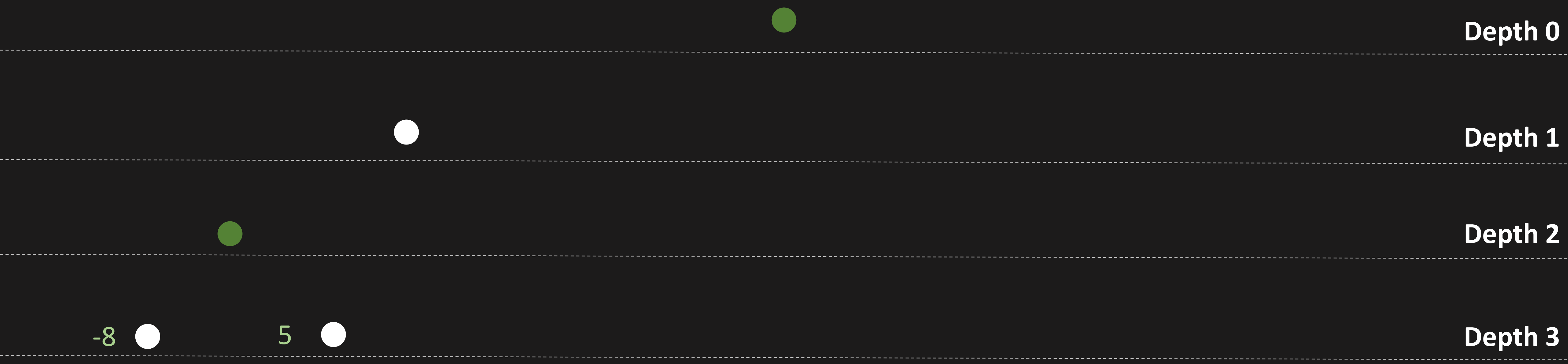
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING

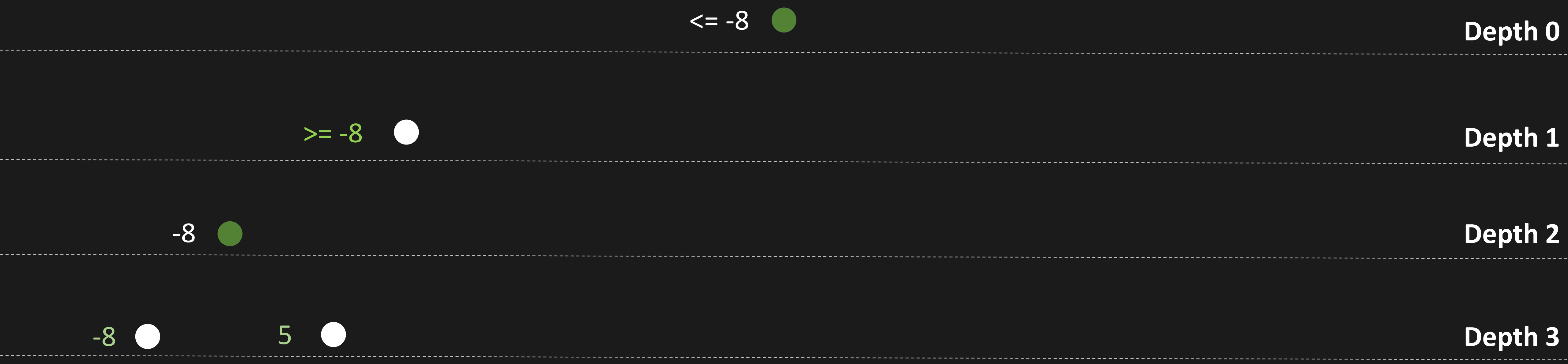


STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING

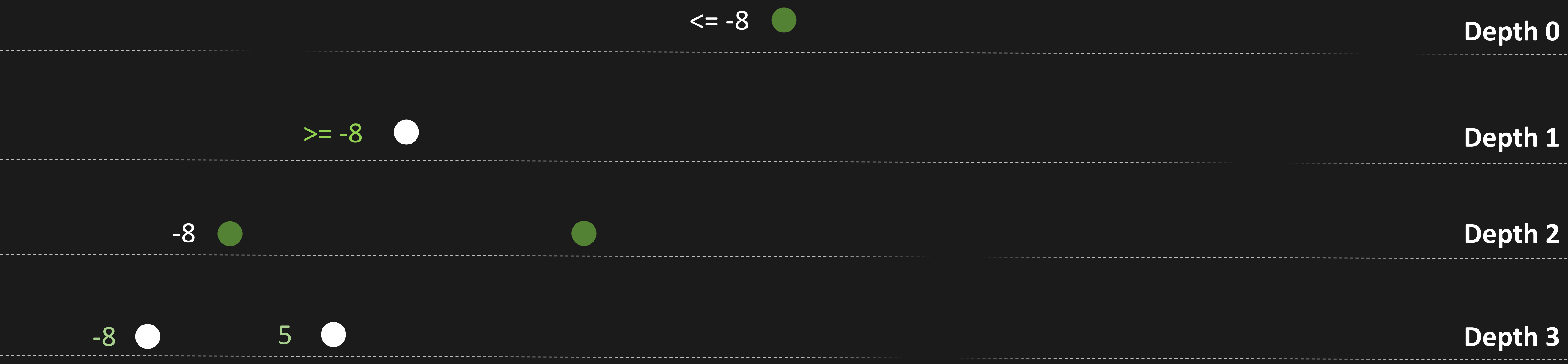


ALPHA BETA PRUNING



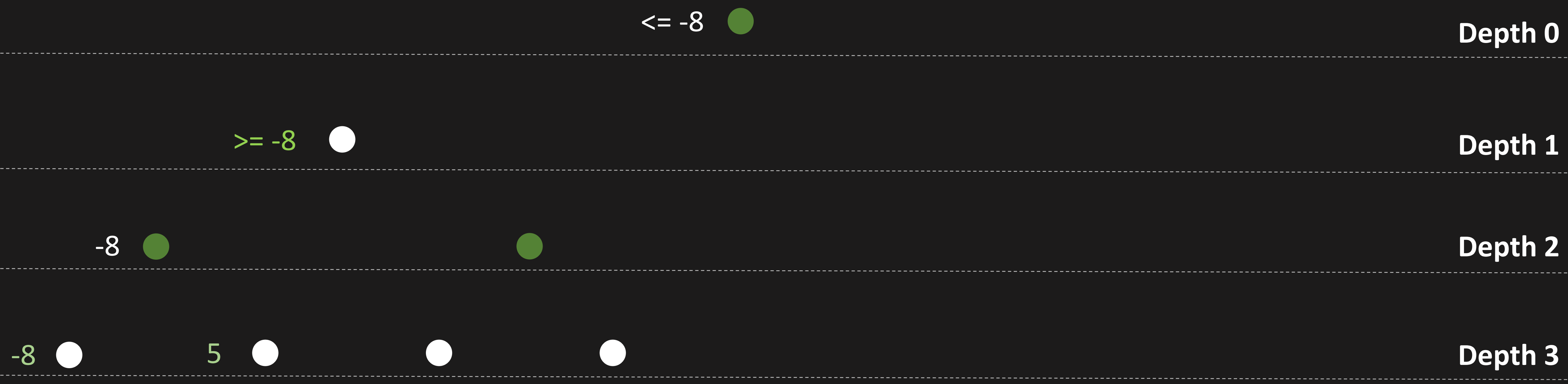
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



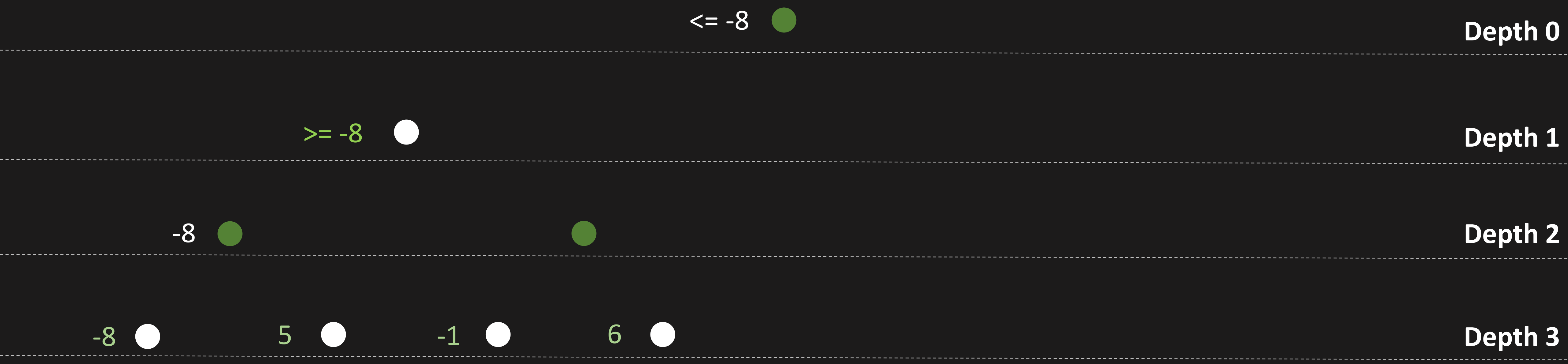
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



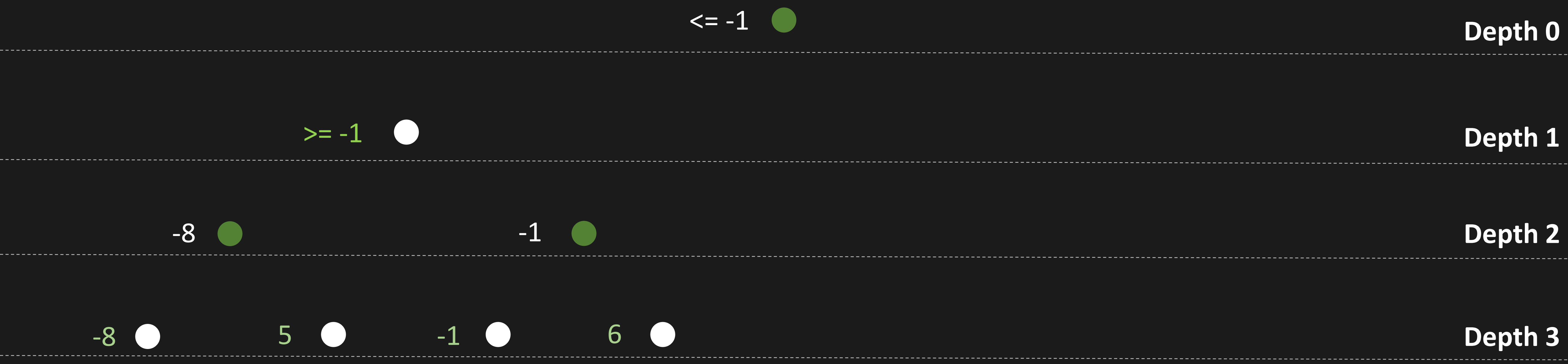
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



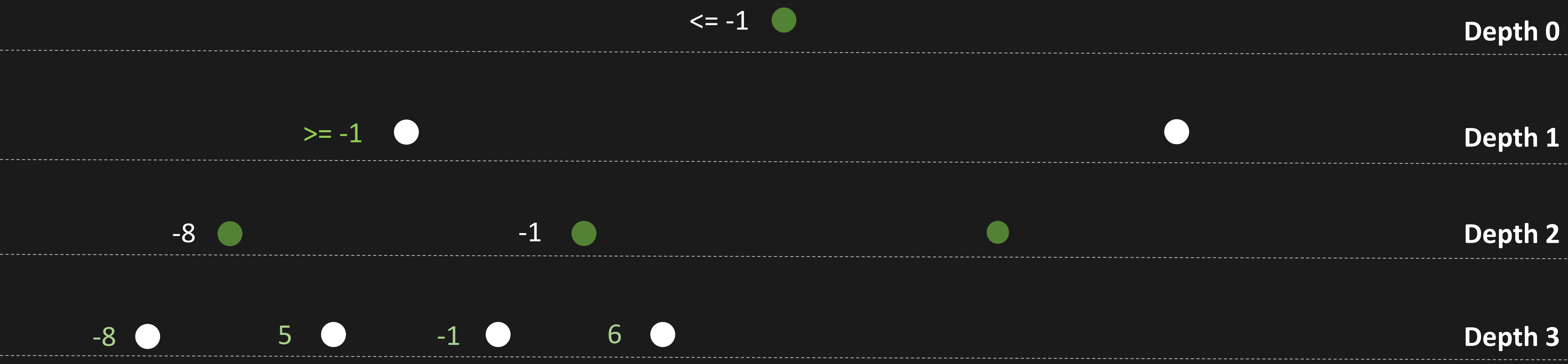
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



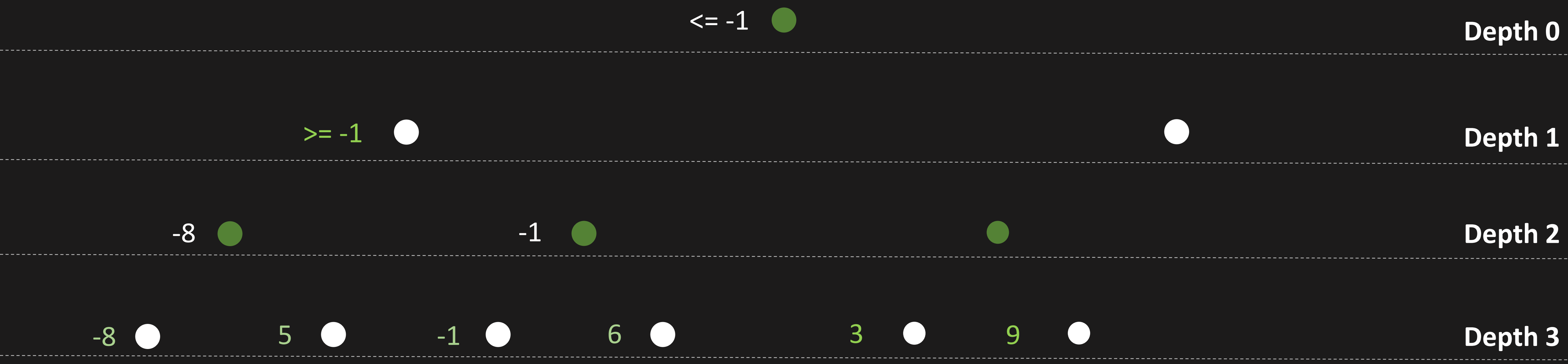
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



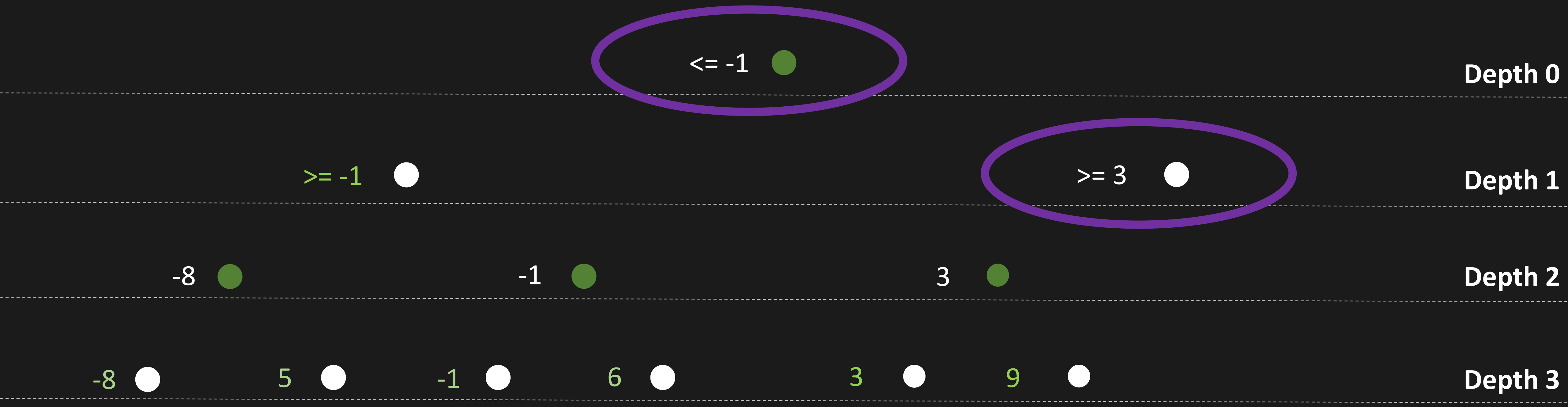
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



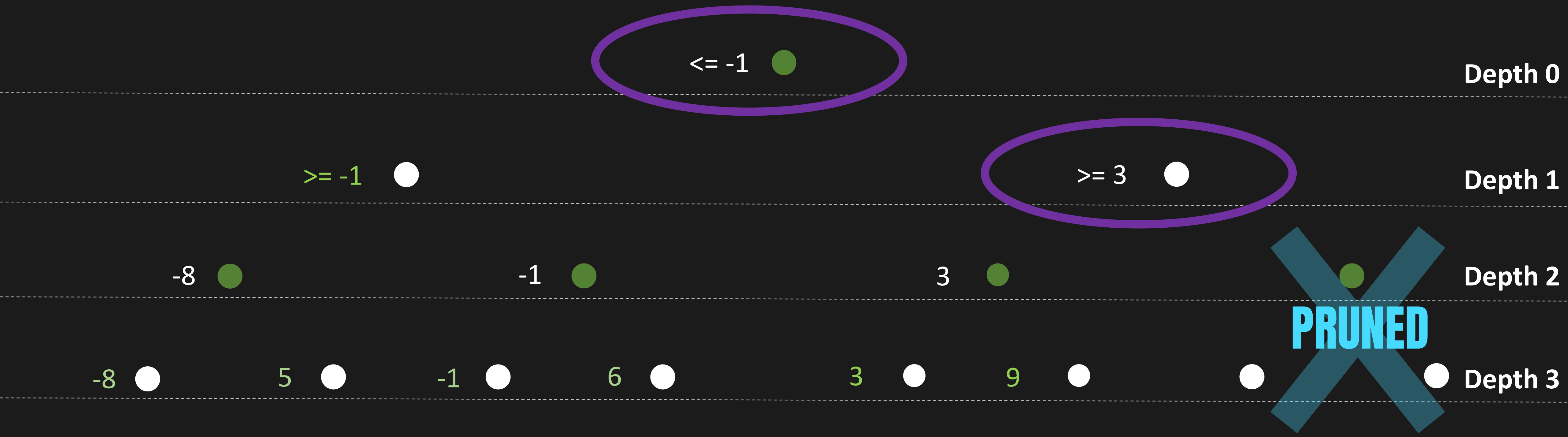
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



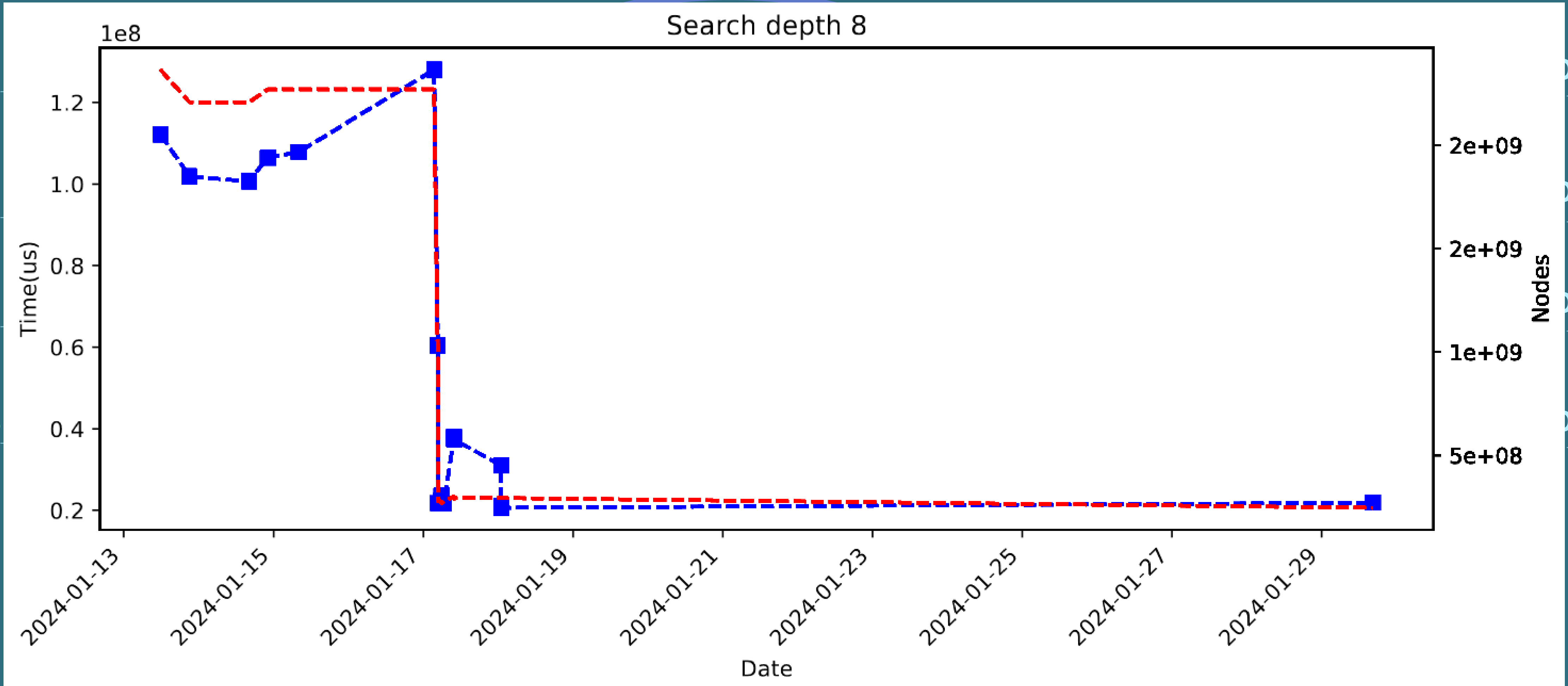
STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computingMay 1978Pages 296–313https://doi.org/10.1145/800133.804359

ALPHA BETA PRUNING



-8

Depth 0

Depth 1

Depth 2

Depth 3

CONCLUSIONS

- Writing a chess engine has been enormously educational.
- The depth of the search is a key factor for the engine strength.
- Optimizing the search algorithm with heuristics can help pruning more branches and allow deeper searches.
- My proto-engine is now online playing other bots
  https://lichess.org/@/ThePaunch
  https://github.com/fraivone/chessengine/tree/v2