Minimax

Team Chairun

Game States

- We can think of a game state as a state in a state space
  - In chess, this refers to a certain board position and the current player to move
- Our state space is the set of all game states
  - Chess is estimated to have 10^{43} possible legal positions
- We can represent the state space as a tree
  - Root: current game state
  - Edge: in-game action that leads from one state to another

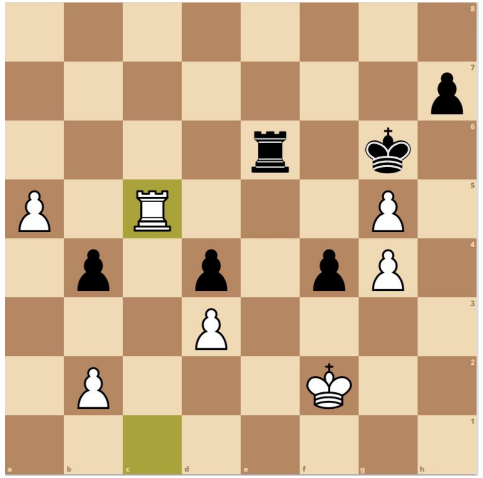


Image from lichess.org

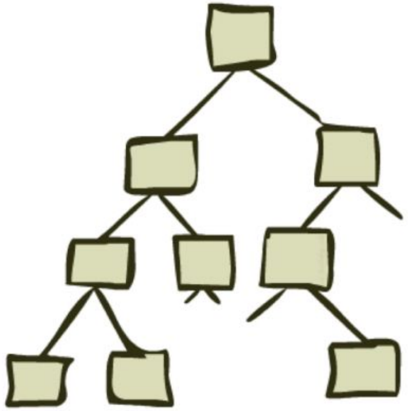
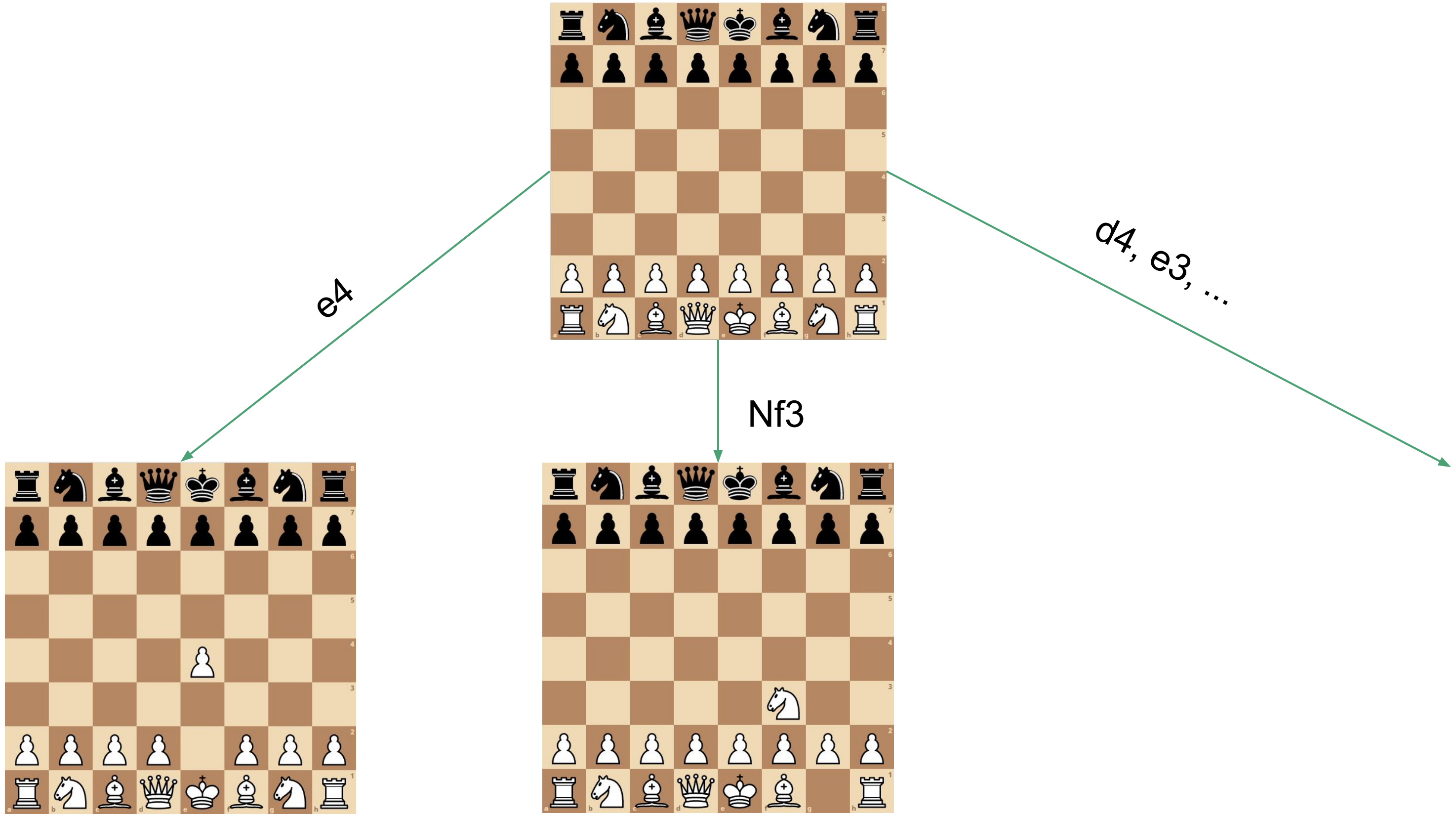


Image from CS188 slides



A depth 1 state space diagram from the starting position (20 legal moves!)

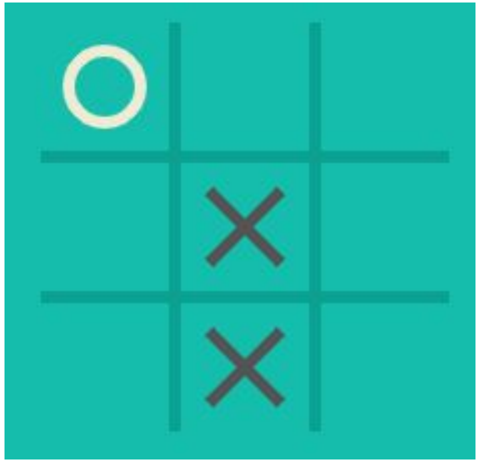
Zero Sum Game

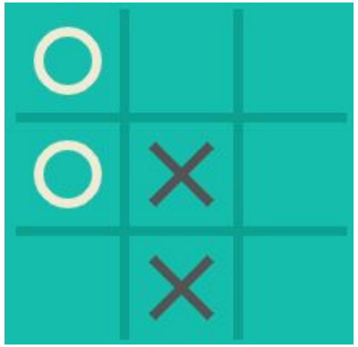
- A zero sum game is “a situation in which each participant's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants” (Wikipedia)
  - e.g. +1 for player A = -1 for player B
- How to find the optimal strategy?
  - Minimax

	r	p	s
r	0	-1	1
p	1	0	-1
s	-1	1	0

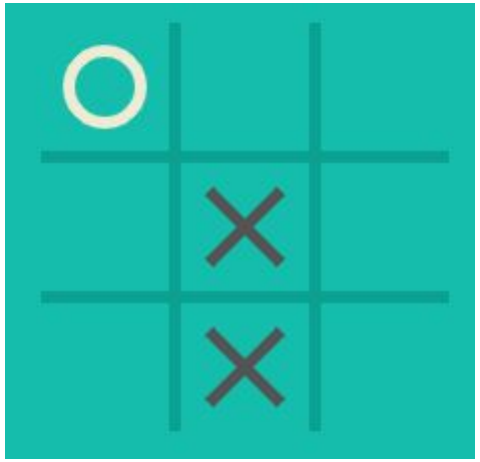
Payoff matrix for rock paper scissors (column player POV)

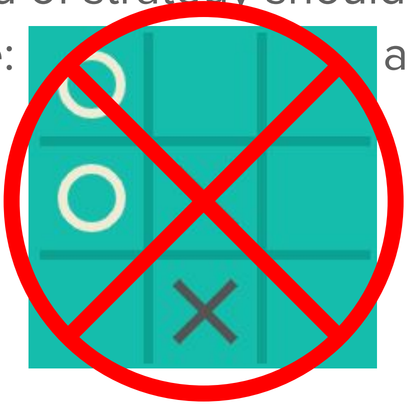
Minimax in Tic-Tac-Toe

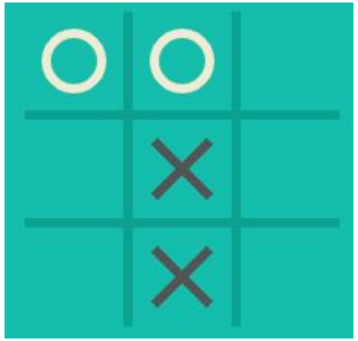


- Suppose the tic-tac-toe game on the right (circle to play):
- What kind of strategy should we take? We could make the move:  and go for a win next move

Minimax in Tic-Tac-Toe

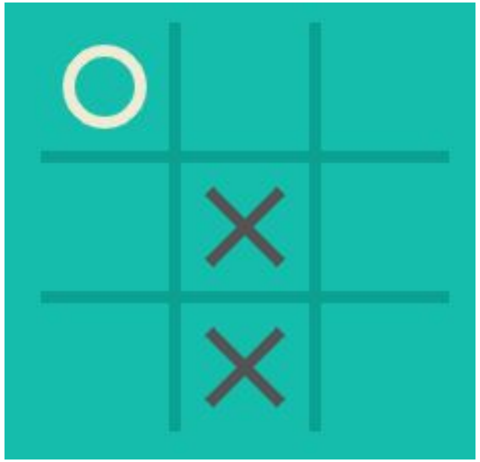


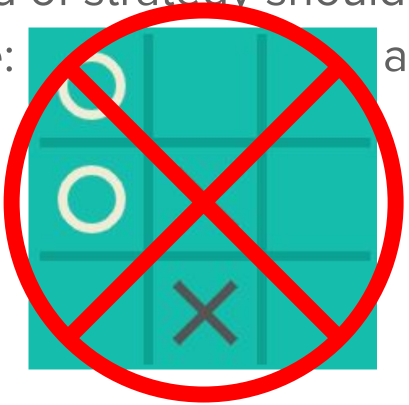
- Suppose the tic-tac-toe game on the right (circle to play):
- What kind of strategy should we take? We could make the move:  and go for a win next move

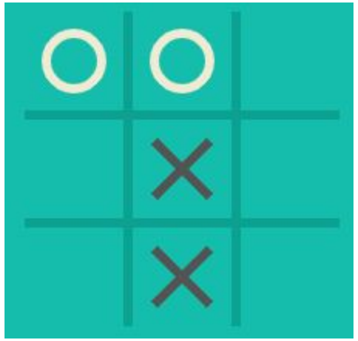


- This strategy is **unsound**,
  as X will win before we have the chance to.
  Hence, we must block X's win:

Minimax in Tic-Tac-Toe



- Suppose the tic-tac-toe game on the right (circle to play):
- What kind of strategy should we take? We could make the move:  and go for a win next move
- This strategy is **unsound**, as X will win before we have the chance to. Hence, we must block X's win:



- We must make the best choice on the assumption that **the opponent will play optimally**
  - Max of my moves (Min of opponent reponses)

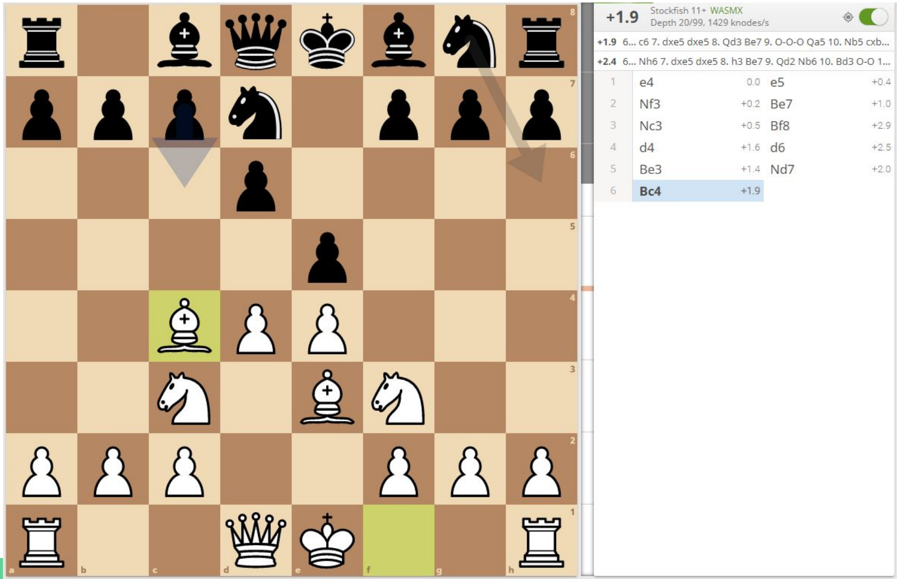
Minimax with a Payoff Matrix

	m	t
e	3	-1
s	-2	1

- Suppose the following payoff matrix:
- The row player announces they will choose either e or s
- Then the column player simply selects the choice that gets the minimum value
- Hence, the row player should pick the choice that maximizes the value the column player will pick
- In this case, max(min(3,-1), min(-2,1)) = -1 so the best strategy for the row player is to pick e and the column player will pick t

Minimax in Chess

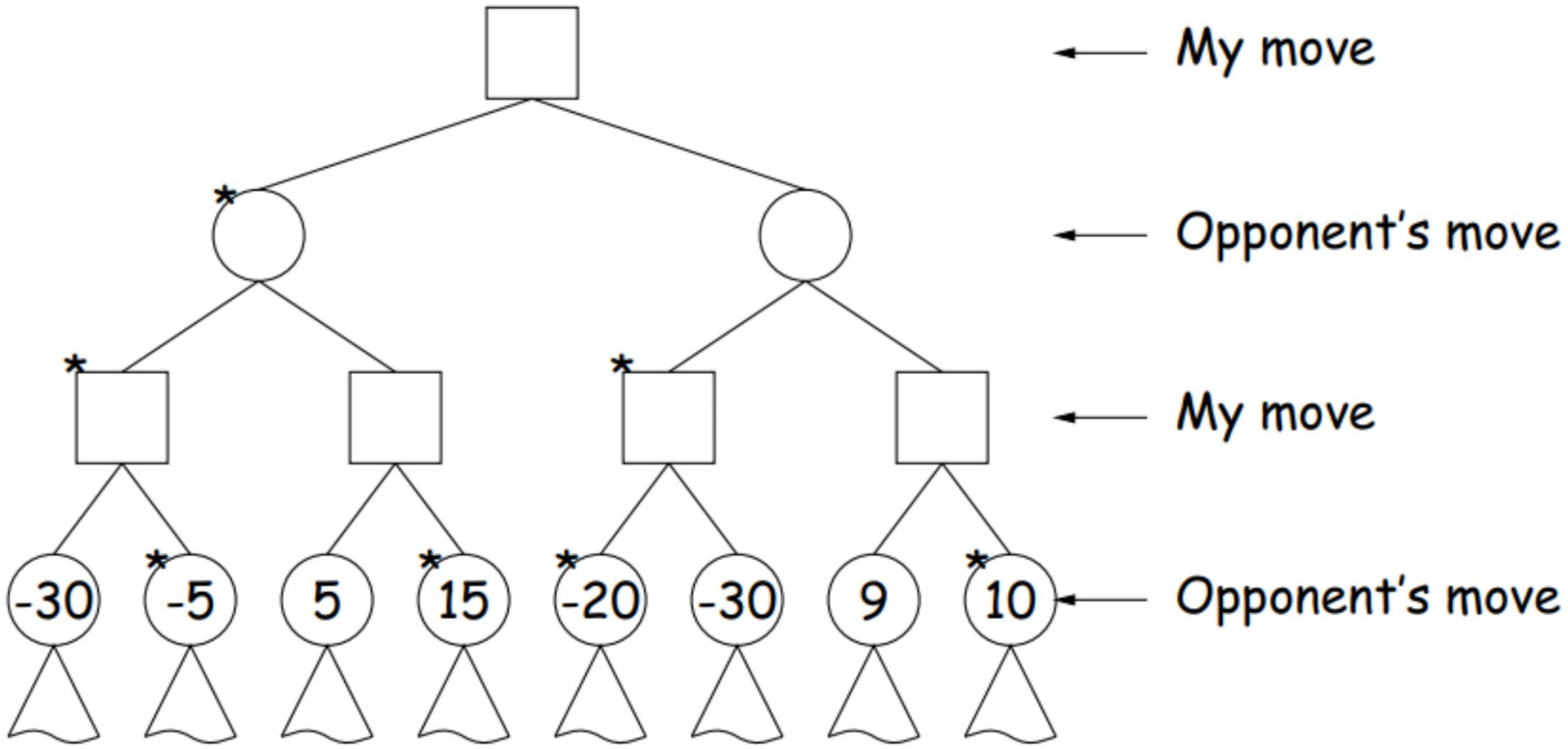
- In tic-tac-toe, we can easily see to the end of the game
- In chess, there are ~20-35 legal moves each turn
  - Hence an exponential growth of states we must consider as we search
- So we don't search to the game's end, but instead stop after a certain number of moves (depth)
- We evaluate the board state using a **heuristic**: a function which generates an assessment of the board
  - Here, stockfish (an engine) determines an advantage of +1.9 pawns for white



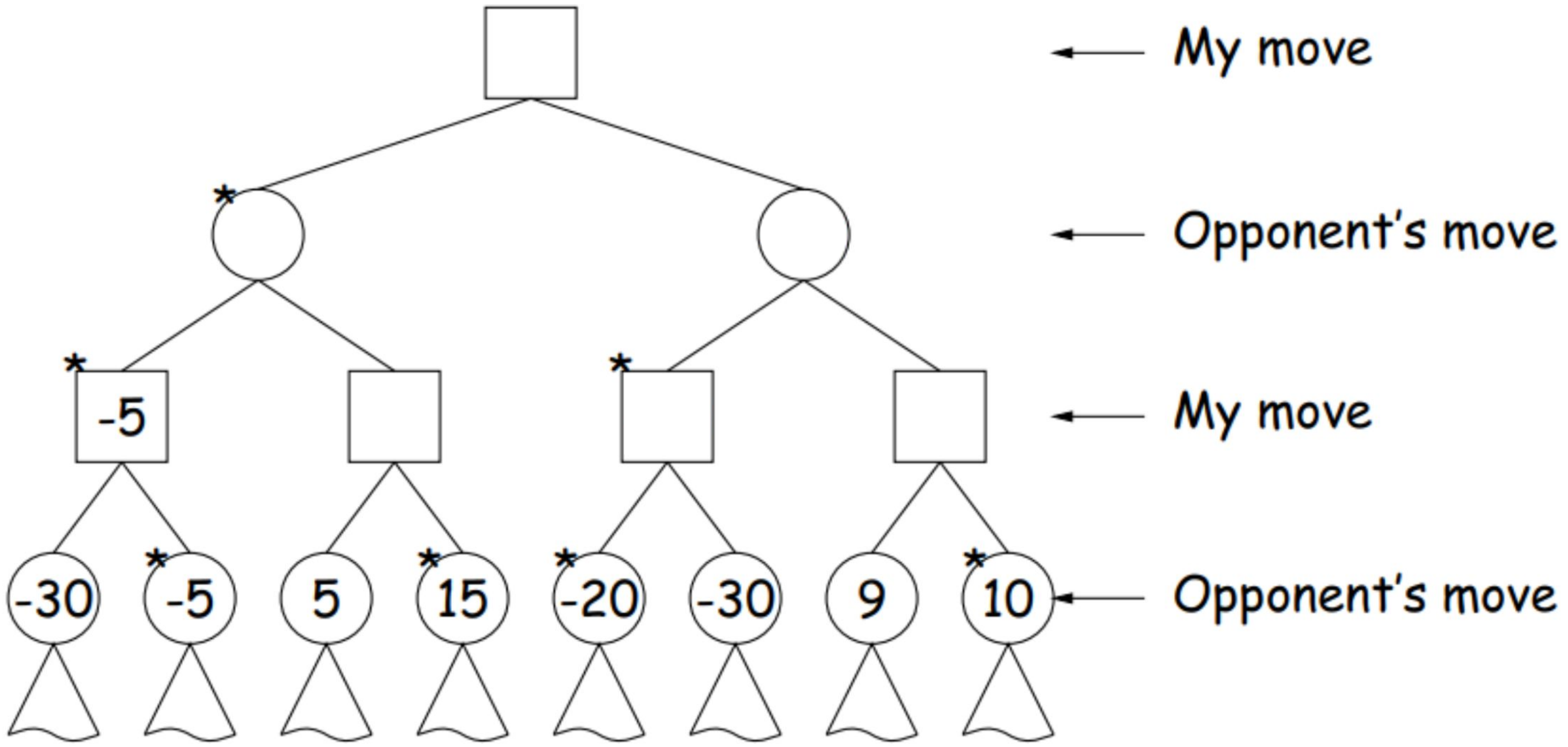
+1.9 Stockfish 11+ WASMX
Depth 20/99, 1429 knodes/s

+1.9 6... c6 7. dxe5 dxe5 8. Qd3 Be7 9. O-O-O Qa5 10. Nb5 cxb...
+2.4 6... Nh6 7. dxe5 dxe5 8. h3 Be7 9. Qd2 Nb6 10. Bd3 O-O 1...

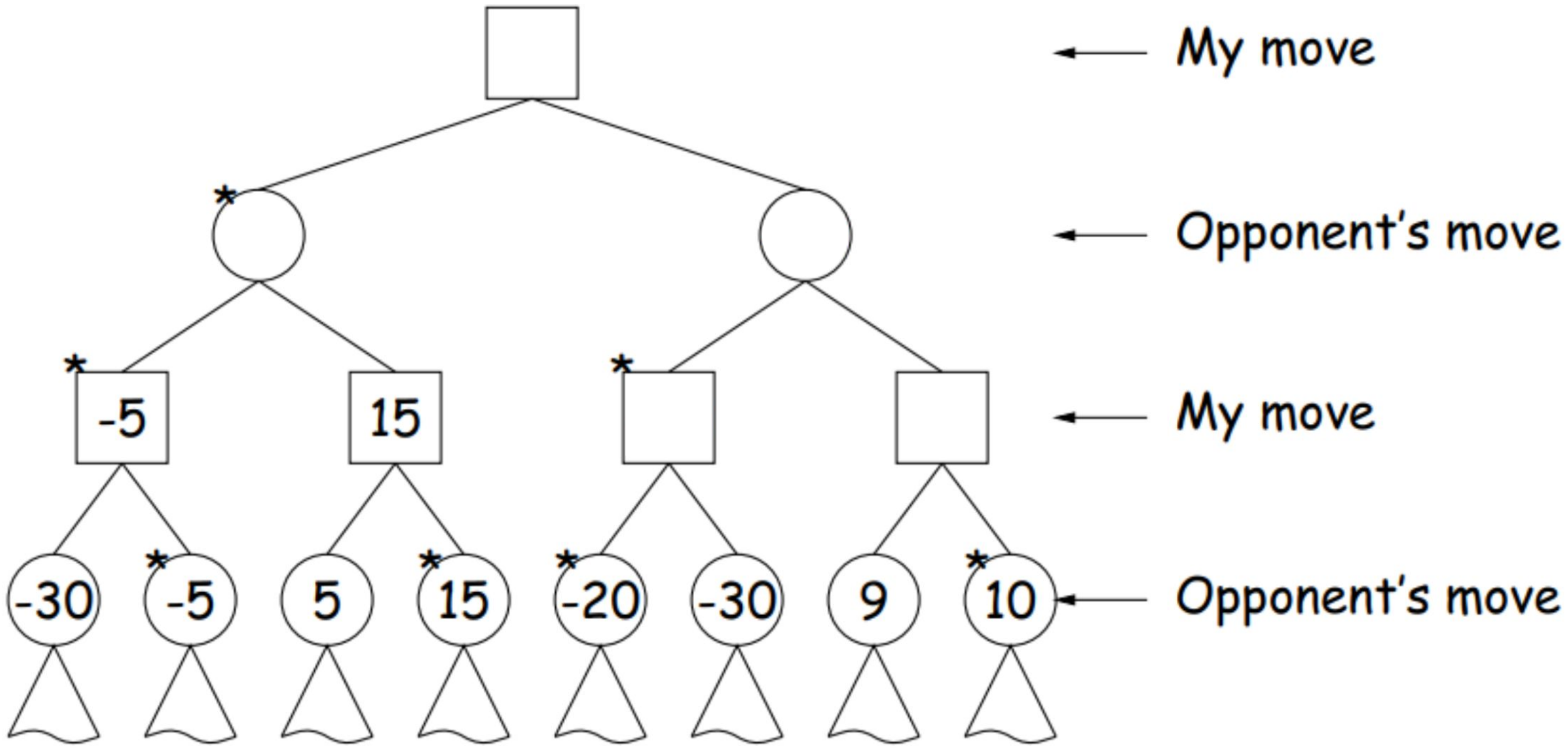
1	e4	0.0	e5	+0.4
2	Nf3	+0.2	Be7	+1.0
3	Nc3	+0.5	Bf8	+2.9
4	d4	+1.6	d6	+2.5
5	Be3	+1.4	Nd7	+2.0
6	**Bc4**	+1.9		



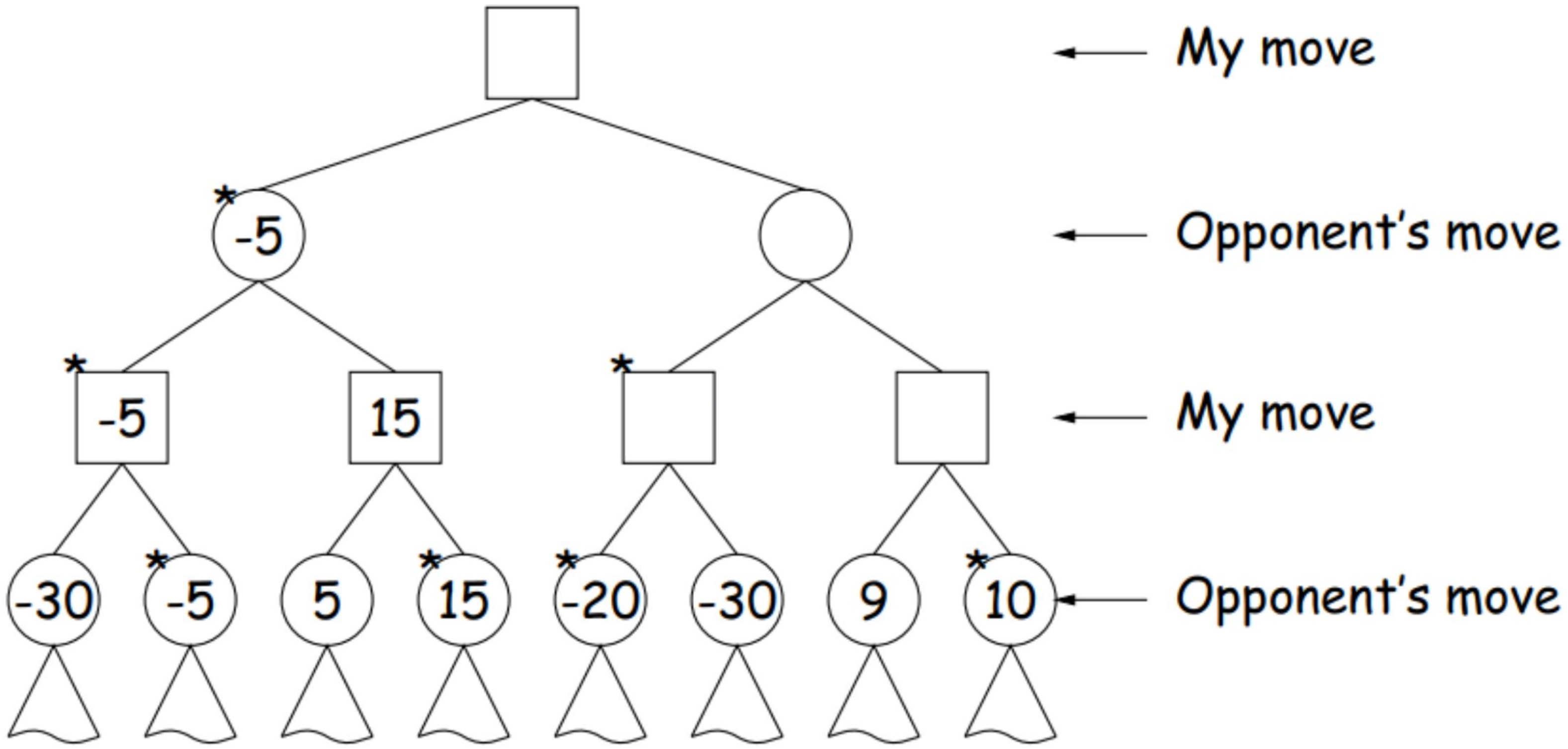
Minimax in a game tree (Image from CS61B slides)



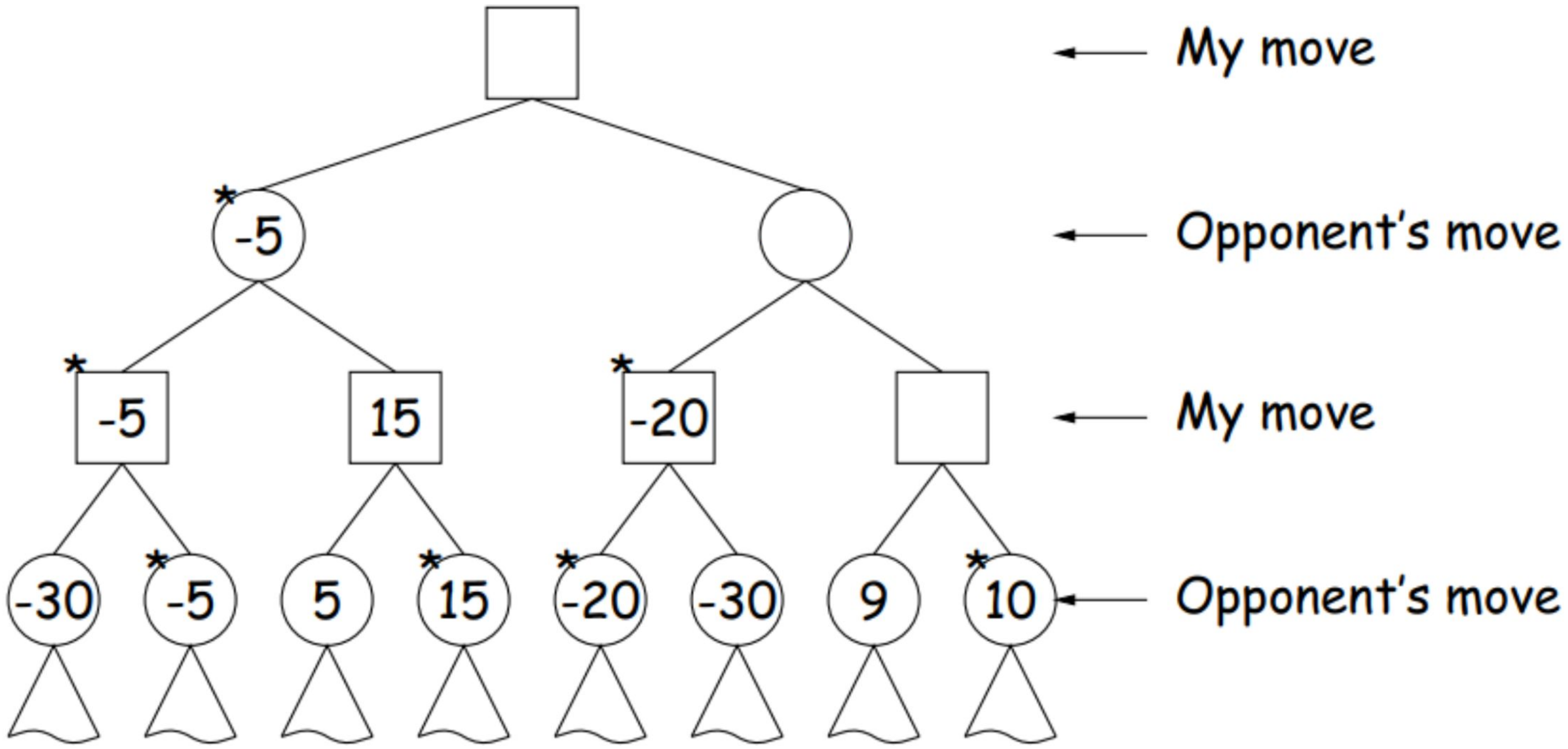
Minimax in a game tree (Image from CS61B slides)



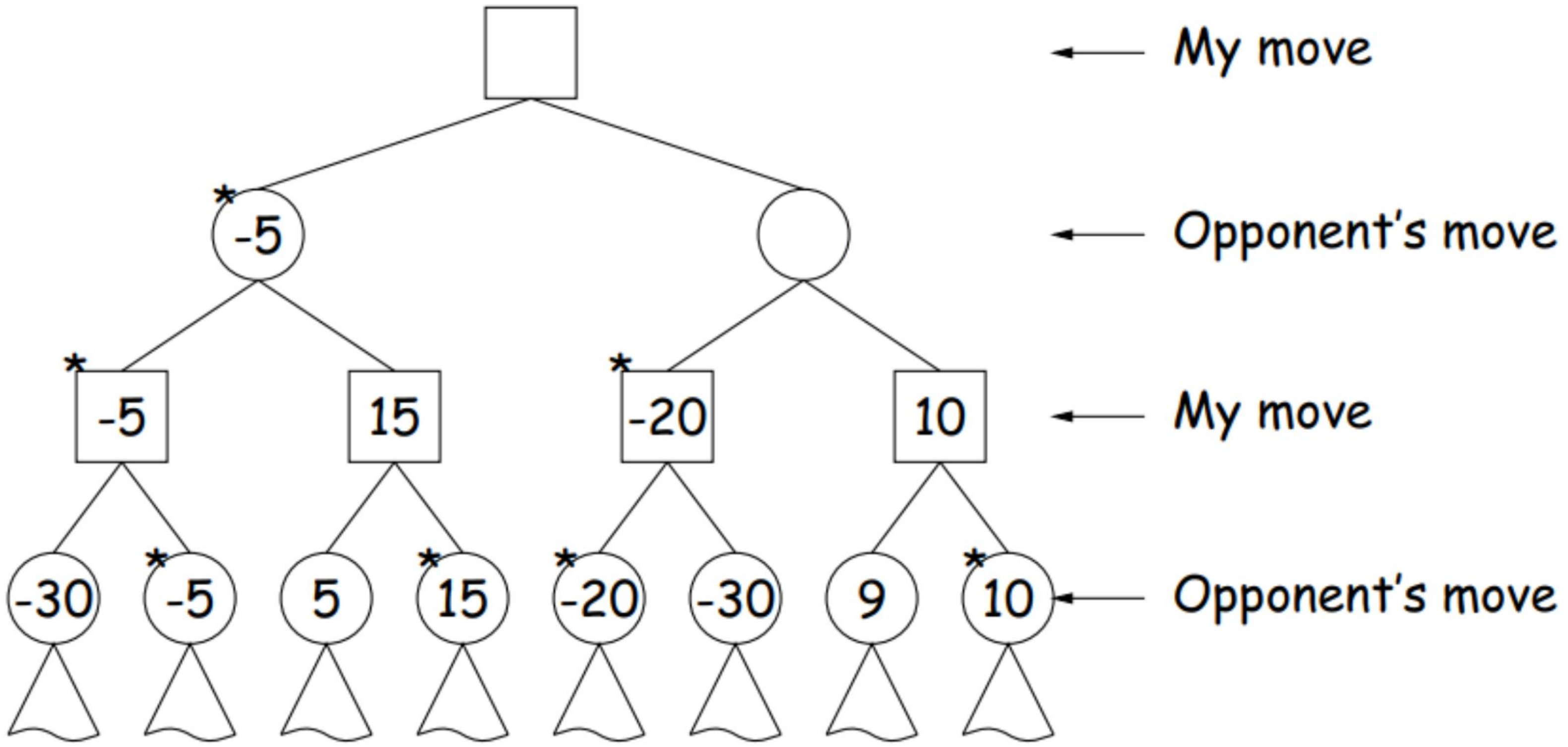
Minimax in a game tree (Image from CS61B slides)



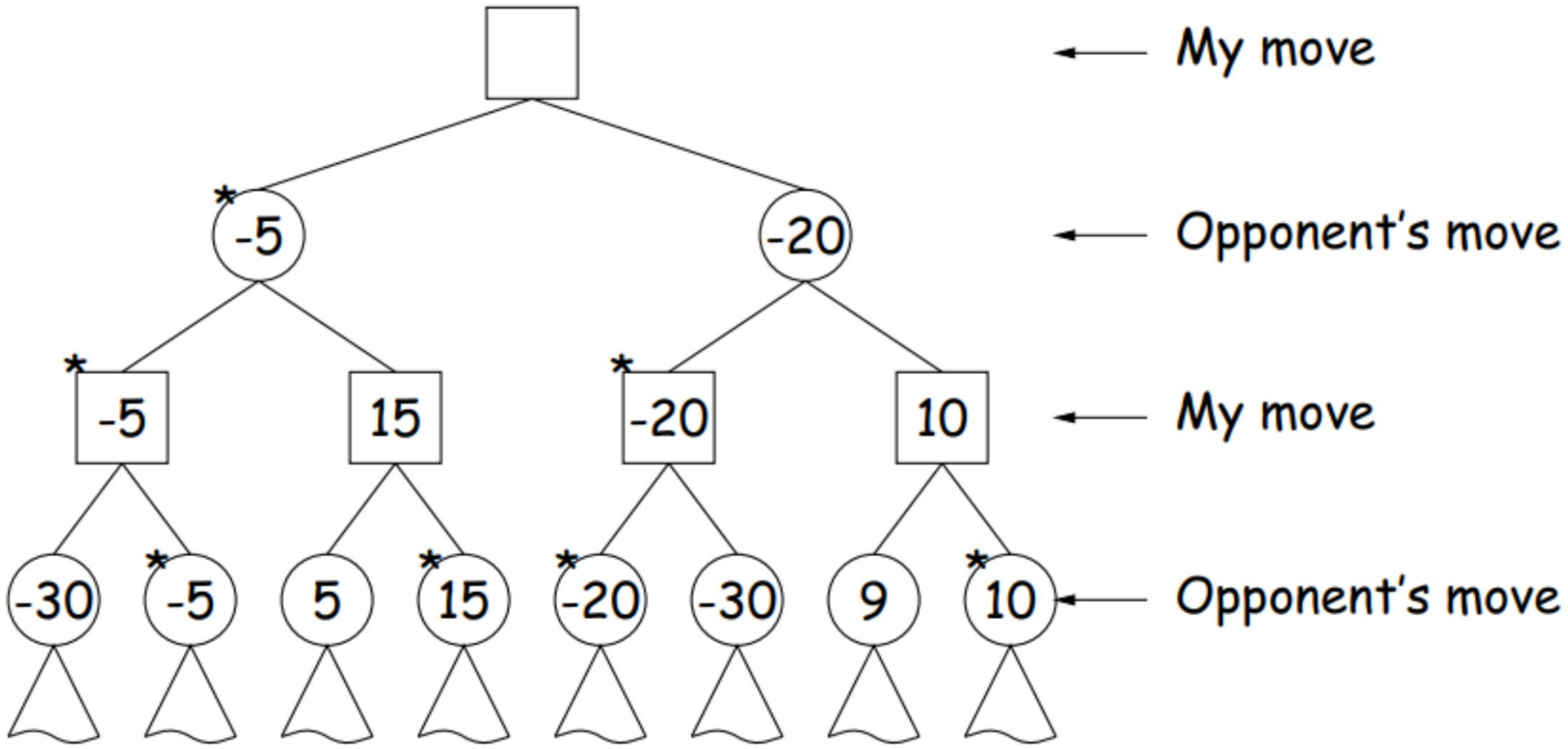
Minimax in a game tree (Image from CS61B slides)



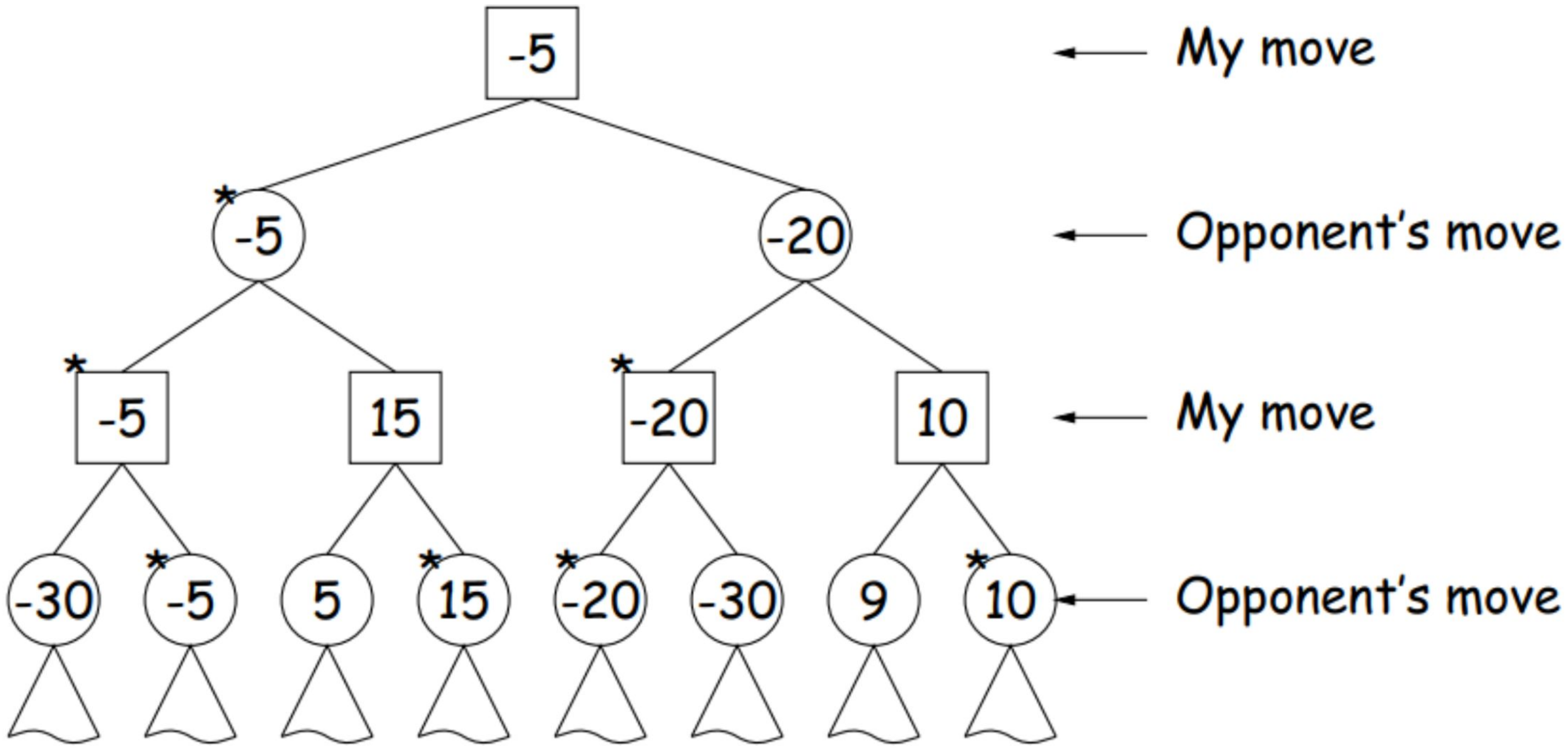
Minimax in a game tree (Image from CS61B slides)



Minimax in a game tree (Image from CS61B slides)



Minimax in a game tree (Image from CS61B slides)



Minimax in a game tree (Image from CS61B slides)

Minimax Pseudocode

```
findMax(position, depth):
    if (depth == 0 || gameOver(position)):
        return heuristic(position)

    best_move_so_far = move with value -infinity
    for(M = legal move):
        position.makeMove(M)
        response = findMin(position, depth-1)
        if(response.val > best_move_so_far.val):
            best_move_so_far = M
    return best_move_so_far
```

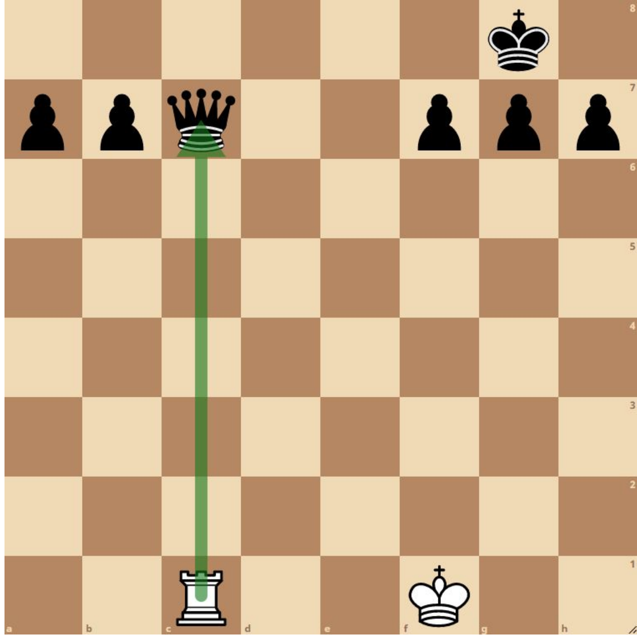
```
findMin(position, depth):
    if (depth == 0 || gameOver(position)):
        return heuristic(position)

    best_move_so_far = move with value +infinity
    for(M = legal move):
        position.makeMove(M)
        response = findMax(position, depth-1)
        if(response.val < best_move_so_far.val):
            best_move_so_far = M
    return best_move_so_far
```

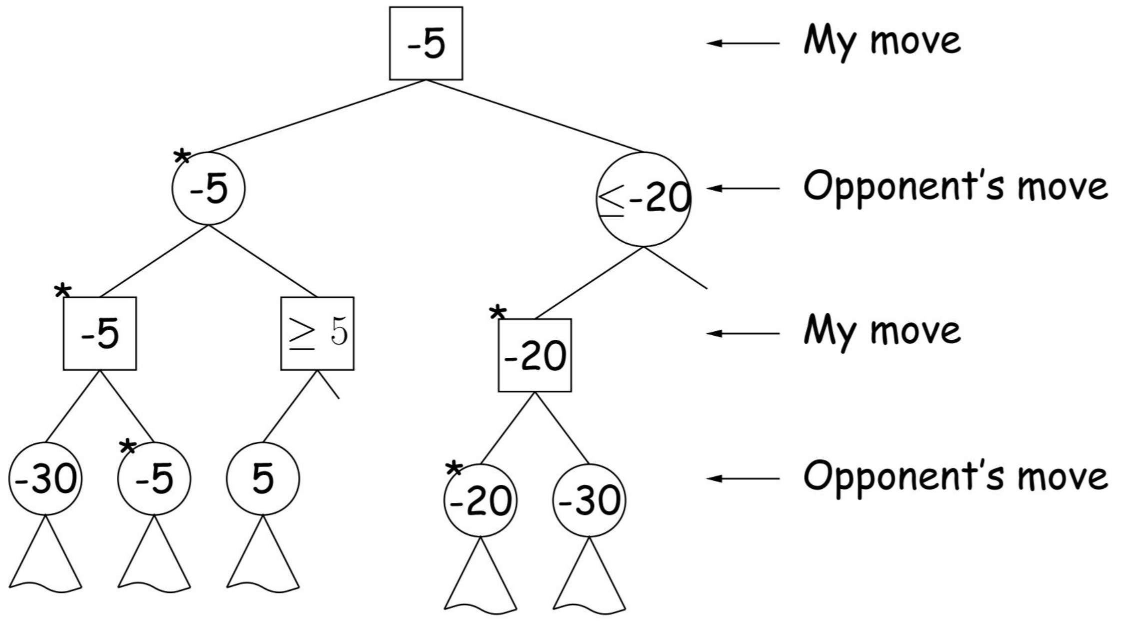
“Twin” functions which call each other: one for min, one for max

With clever sign manipulation, we can combine the two into one function

Search Redundancy



- Suppose the position on the right (white to move)
- Analysis as a human:
  - Rxc7 is absurdly good (winning a Queen for free)
  - Hence all other moves are **pointless to search**
- Analysis as a machine:
  - Rxc7, leads to +2.5
  - **Try all other possible moves, try all possible responses**, …
    - Leads to -70 or so
- As a human, we recognize that it is pointless to look at other moves **since we already have found a superior move**
  - The instant we see moving the king **could** lead to some worse value, it’s not productive to look further - we already have something better!
  - For a machine, this is called **pruning**



- At the '≥ 5' position, I know that the opponent will not choose to move here (since he already has a -5 move).
- At the '≤ -20' position, my opponent knows that I will never choose to move here (since I already have a -5 move).

Pruning in a game tree (Image and text from CS61B slides)

Alpha-Beta Pruning

```
findMax(position, depth, alpha, beta):
    if (depth == 0 || gameOver(position)):
        return heuristic(position)

    best_move_so_far = move with value -infinity
    for(M = legal move):
        position.makeMove(M)
        response = findMin(position, depth-1,
                              alpha, beta)
        if(response.val > best_move_so_far.val):
            best_move_so_far = M
            alpha = max(alpha, M.val)
            if (beta <= alpha): break
    return best_move_so_far
```

```
findMin(position, depth, alpha, beta):
    if (depth == 0 || gameOver(position)):
        return heuristic(position)

    best_move_so_far = move with value +infinity
    for(M = legal move):
        position.makeMove(M)
        response = findMax(position, depth-1,
                              alpha, beta)
        if(response.val < best_move_so_far.val):
            best_move_so_far = M
            beta = min(beta, M.val)
            if (beta <= alpha): break
    return best_move_so_far
```

Each position carries a upper bound beta and lower bound alpha:

alpha = the worst (lowest) result the max player would accept

beta = the best (highest) result the min player would accept

What About Time?

- Currently, our search is run with fixed depth - it’s difficult to estimate of how long this will take
- In games like chess, there are limitations on the amount of time used by each player
  - Selecting depth is not a realiable way to manage time
- How about stopping when time runs out?
  - When we begin a search of depth d, we must finish the search
  - A search done halfway is **ineffective** - the next move we look at could be the best one

Chess clock. Image from Wikipedia.

Iterative Deepening

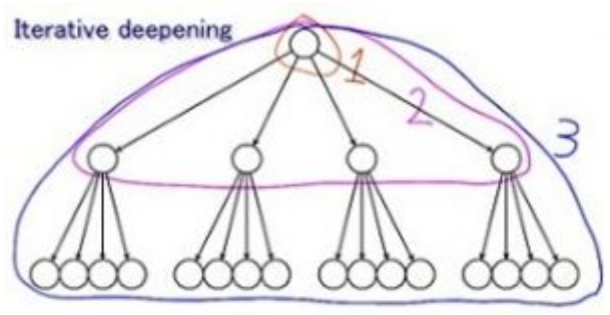


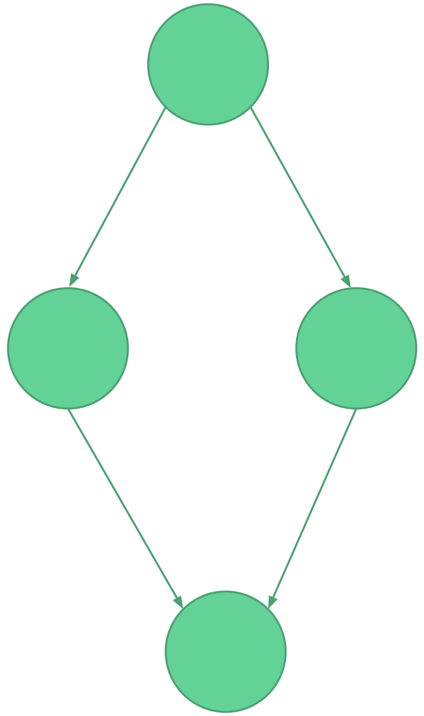
Image from chess programming wiki

- A search of depth d is strictly less accurate than a search of depth d+1
    - So what if we increased the depth until we ran out of time?
- This is called iterative deepening: we start at depth = 1, then run depth = 2…
    - If we run out of time on depth d, we return the result of depth d-1
- Why not use breadth-first-search instead to reduce redundancy?
    - BFS requires too much memory: need to store the bottom layer at every step (which can be extremely large at higher depths)
    - The asymptotic runtime is the same, but BFS has much worse memory usage

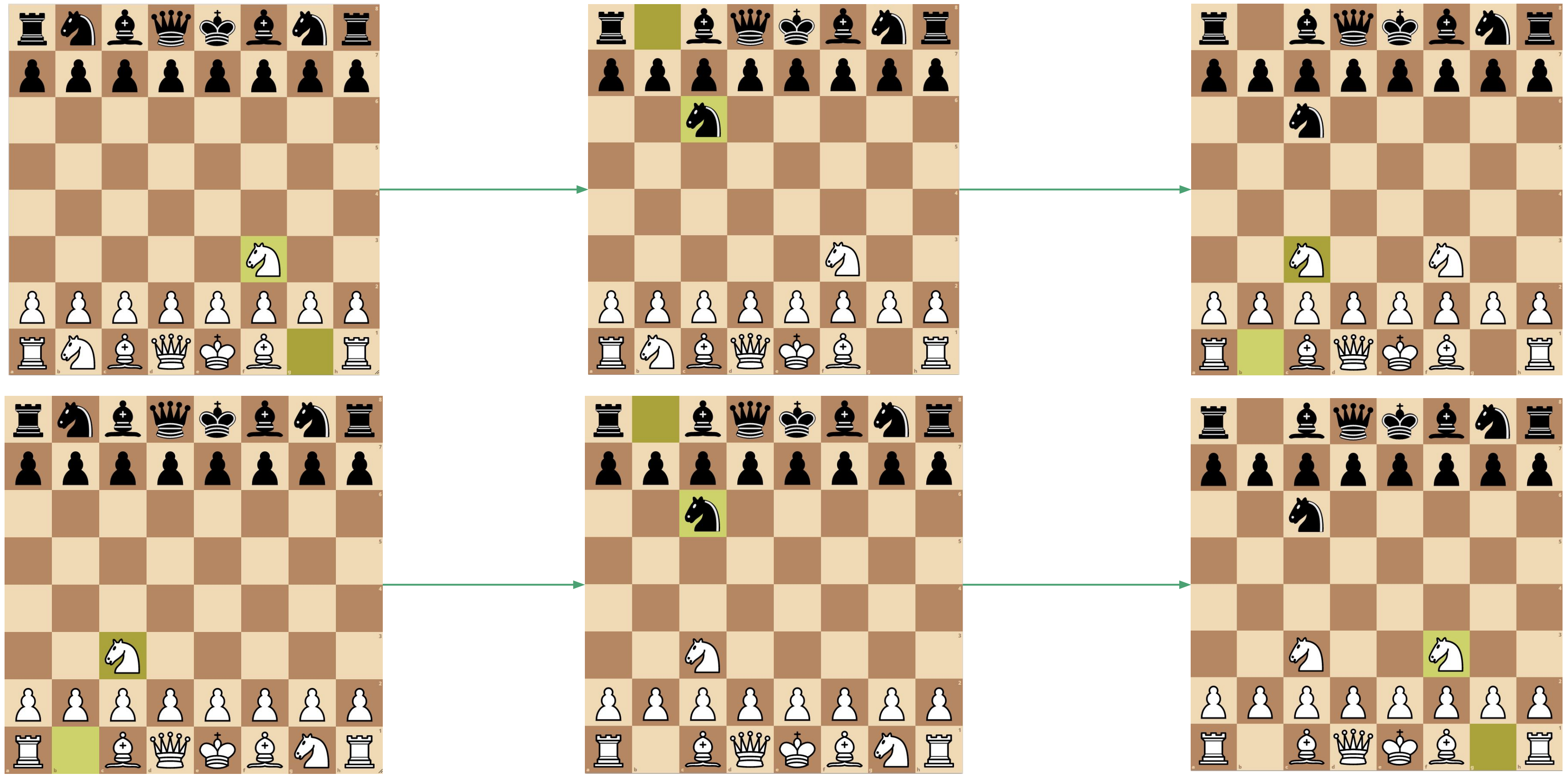
Advanced Extensions to Minimax

Transposition

- In games such as chess, our evaluation is **memoryless**: the evaluation is dependent on the current position and independent of the previous moves
- We want to ensure that **regardless of move order, the same position gives the same output**
    - We don't need to search this position again, so we can save computation
- Two move orderings that give the same position are called the same by **transposition**

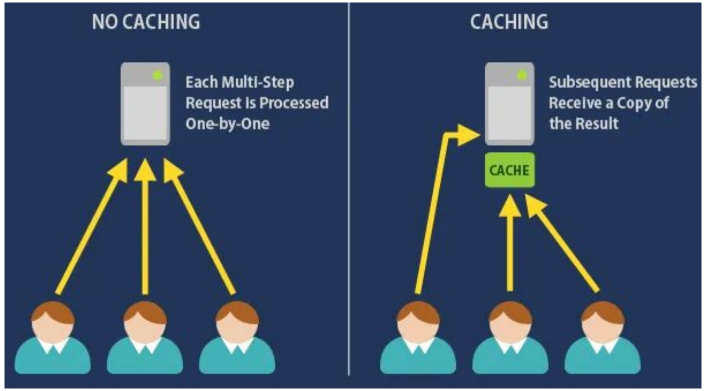


The two paths are the same by transposition


1. Nf3 Nc6 2. Nc3 (top) and 1. Nc3 Nc6 2. Nf3 (bottom) are the same by transposition

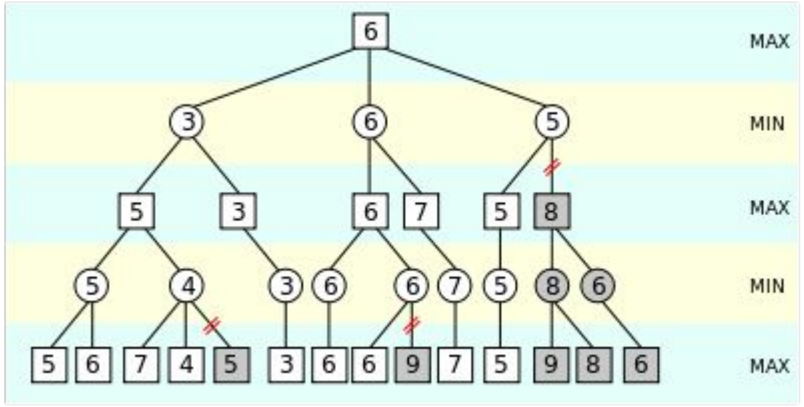
Transposition Table

- To exploit this property, we will store positions we have already seen in a **transposition table**
    - Then we can return the results of previous searches instead of searching again
- However, we can't save each position (since there are typically too many), so we only store a set amount
- We can look up positions we see in our table using a **hasher**
    - For more details, take CS61B or CS170
- This effectively serves as a **cache** for our searches
    - There are various ways to replace entries in this table covered in the note


NO CACHING
Each Multi-Step Request Is Processed One-by-One
CACHING
Subsequent Requests Receive a Copy of the Result
CACHE

Caching. Image from liquidweb.com

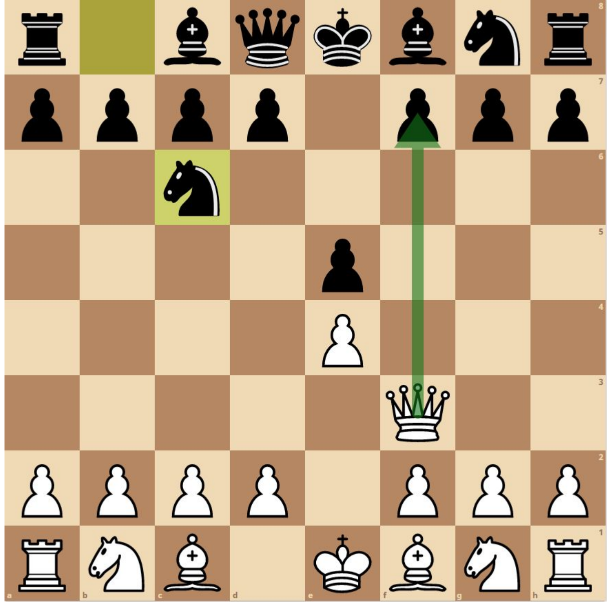
Move Ordering



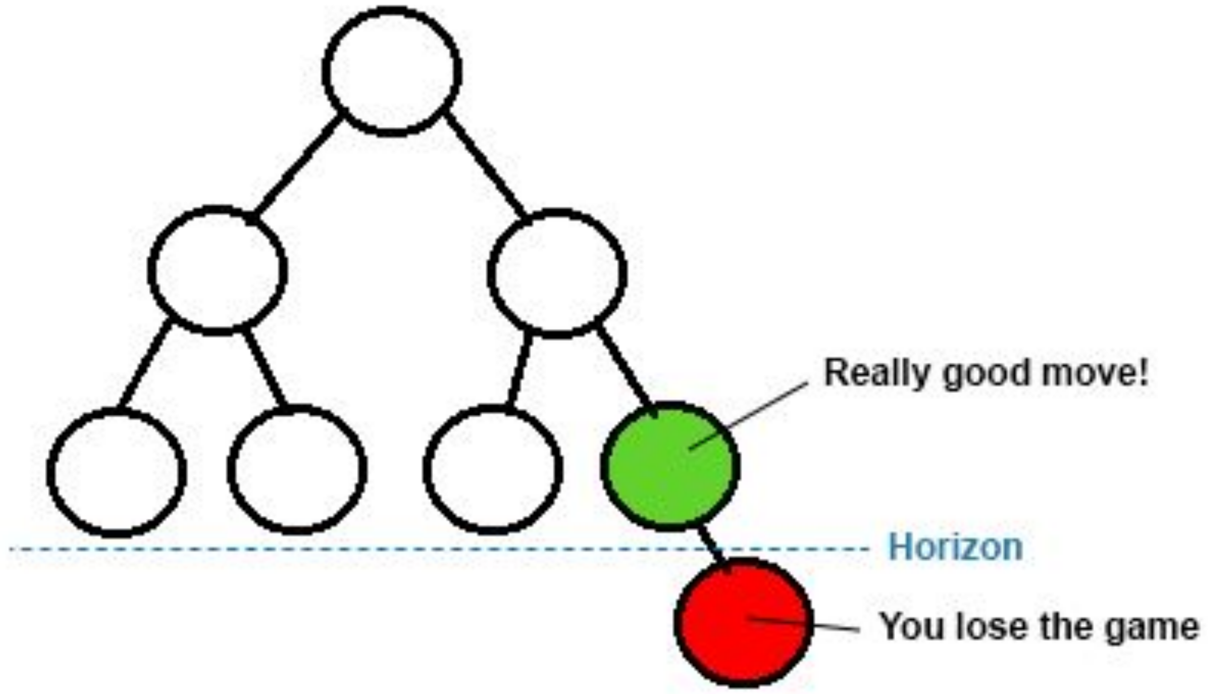
If we had searched the branch "6" first, we could have pruned the branch "3" much faster

- Suppose we run alpha-beta pruning on moves {a, b, c}, where a > b > c, with the goal to maximize
  - Search order c, b, a: we will be unable to prune any move, since the next move is better than the previous one
  - Search order a, b, c: we will likely be able to prune b and c, since we already know that a is better
- Hence, it is always best to **search the best moves first** since you will be able to prune the worse moves
  - We can't know in advance, but we can estimate which moves look the most promising

Horizon Effect



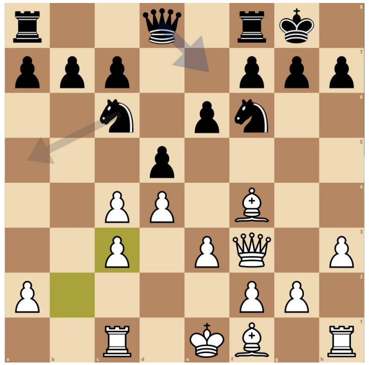
- Suppose we search the position on the right at depth 1, and we see Qxf7
  - Our AI will stop the search after this move, and see we have won a pawn!!
  - The AI doesn't see that the next turn we lose our Queen
- This is called the **horizon effect**: we are unable to see the consequences of our bad moves since their effects are at a higher depth (or beyond the horizon)
- How can we combat this?

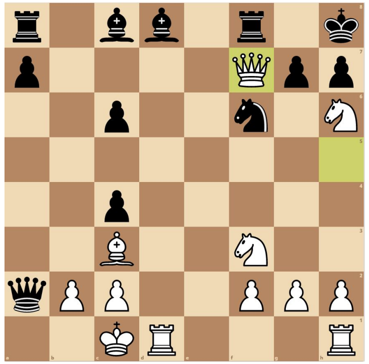


Tree perspective of the Horizon effect. Image from krystman.itch.io

Quiescence Search and Quiet Positions

- **Quiescence search** is an extension of minimax - when we hit the end of minimax, we continue searching on a limited set of moves
- The idea is we want to stop only when the position is **quiet**, or non-volatile
    - Our current evaluation should be a good estimate of the true value, even at higher depths
- In chess, we might want to search for only checks and captures
    - Typically, these cause the largest swings in evaluation

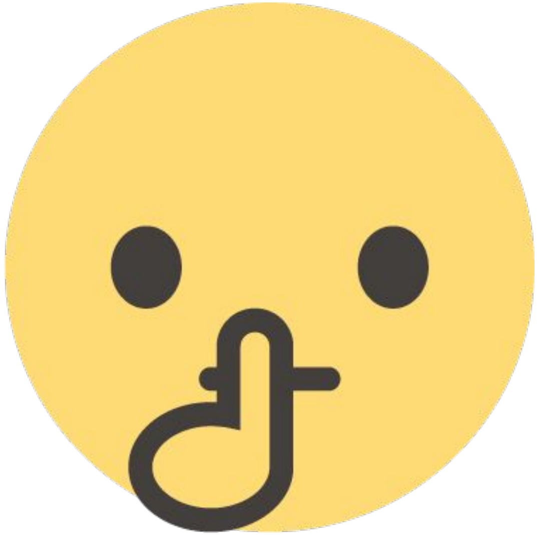

A quiet position: no good moves will cause large swings


A non-quiet position: both players are getting checkmated

Quiescence Search Pseudocode

```
function search(node, depth):
    if depth == 0:
        return quiescence_search(node, reasonable_depth_value)
    for all moves:
        search(next_node, depth - 1)
    return value of children

function quiescensce_search(node, depth):
    If depth == 0 or node appears quiet:
        Return estimated value of node
    for all moves:
        quiescence_search(next_node, depth - 1)
    return value of children
```



Search if the position is not quiet.
Image from iconbros.com

References

- CS189 course slides
- CS61B course slides
- Chessprogramming wiki
- Wikipedia
- lichess.org