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

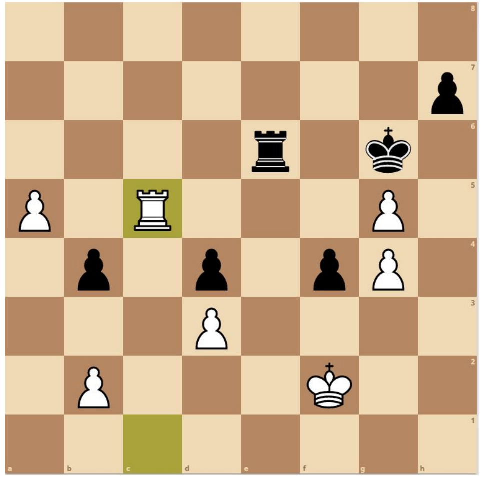


Image from lichess.org

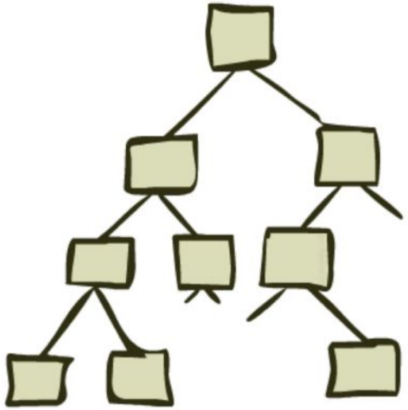
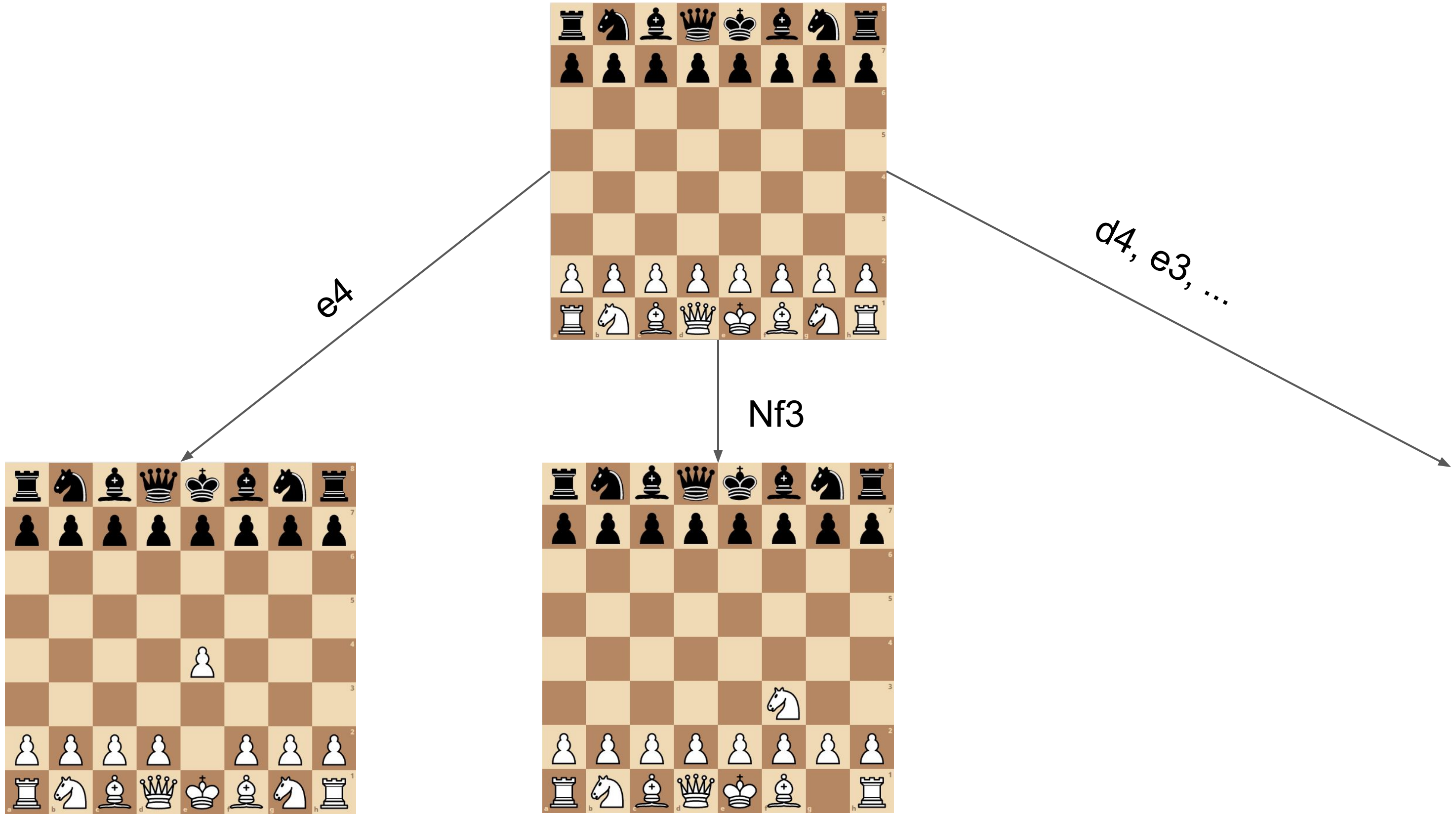


Image from CS188 slides



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

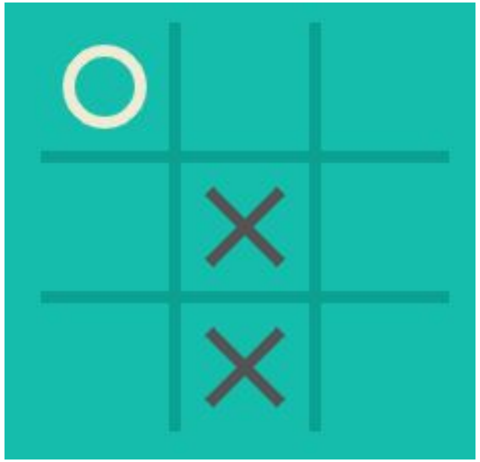
Payoff matrix for rock paper scissors (column player POV)

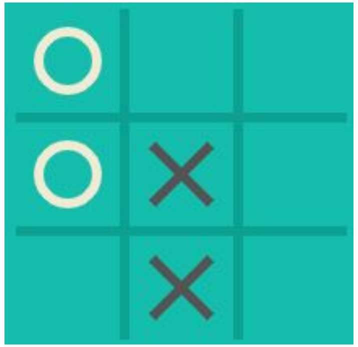
Minimax with a Payoff Matrix

	m	t
e	3	-1
s	-2	1

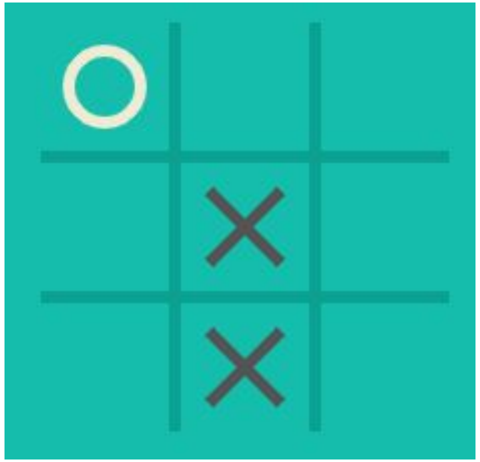
- Suppose the following payoff matrix:
- The row player announces a strategy to choose either e or s
- Then the column player simply chooses the strategy that gets the minimum value
- The row player should pick the strategy that maximizes the value the column player will pick
- In this case, max(min(3,-1),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

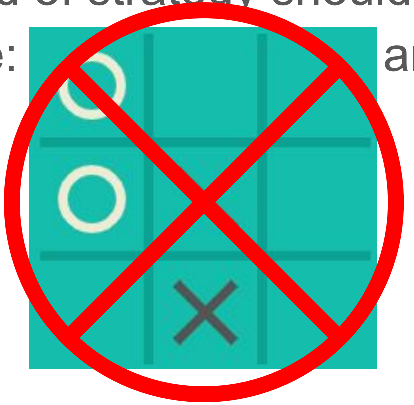
Minimax in Tic-Tac-Toe

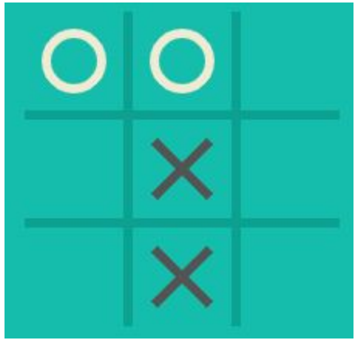


- Suppose the tic-tac-toe game on the right (circle to play):
- What kind of strategy should we take? We could make the move:  and go for a win next move

Minimax in Tic-Tac-Toe

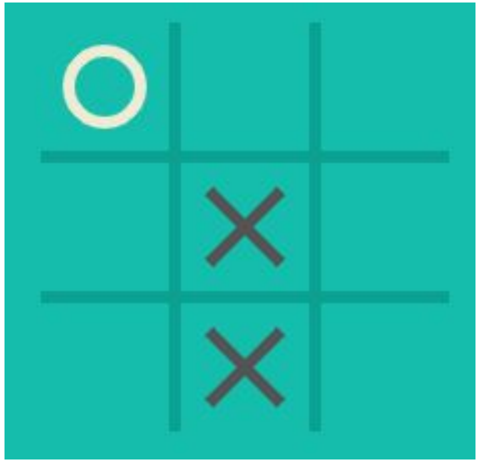


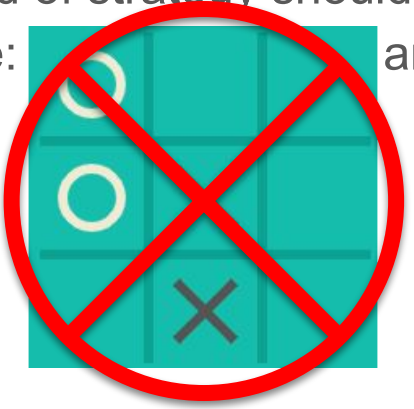
- Suppose the tic-tac-toe game on the right (circle to play):
- What kind of strategy should we take? We could make the move:  and go for a win next move

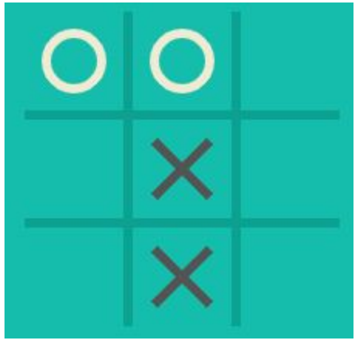


- This strategy is **unsound**, as X will win before we have the chance to. Hence, we must block X's win:

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

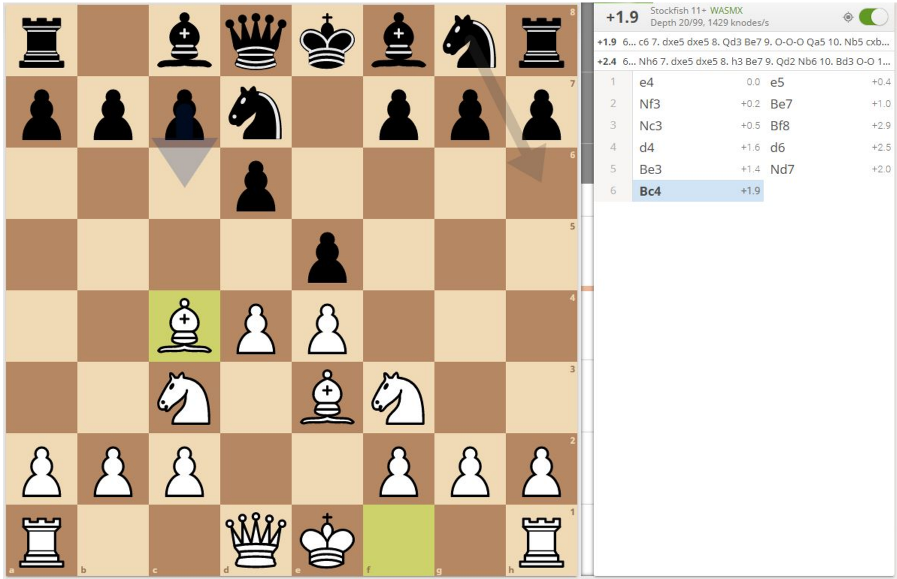


- This strategy is **unsound**, as X will win before we have the chance to. Hence, we must block X's win:

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

Minimax in Chess

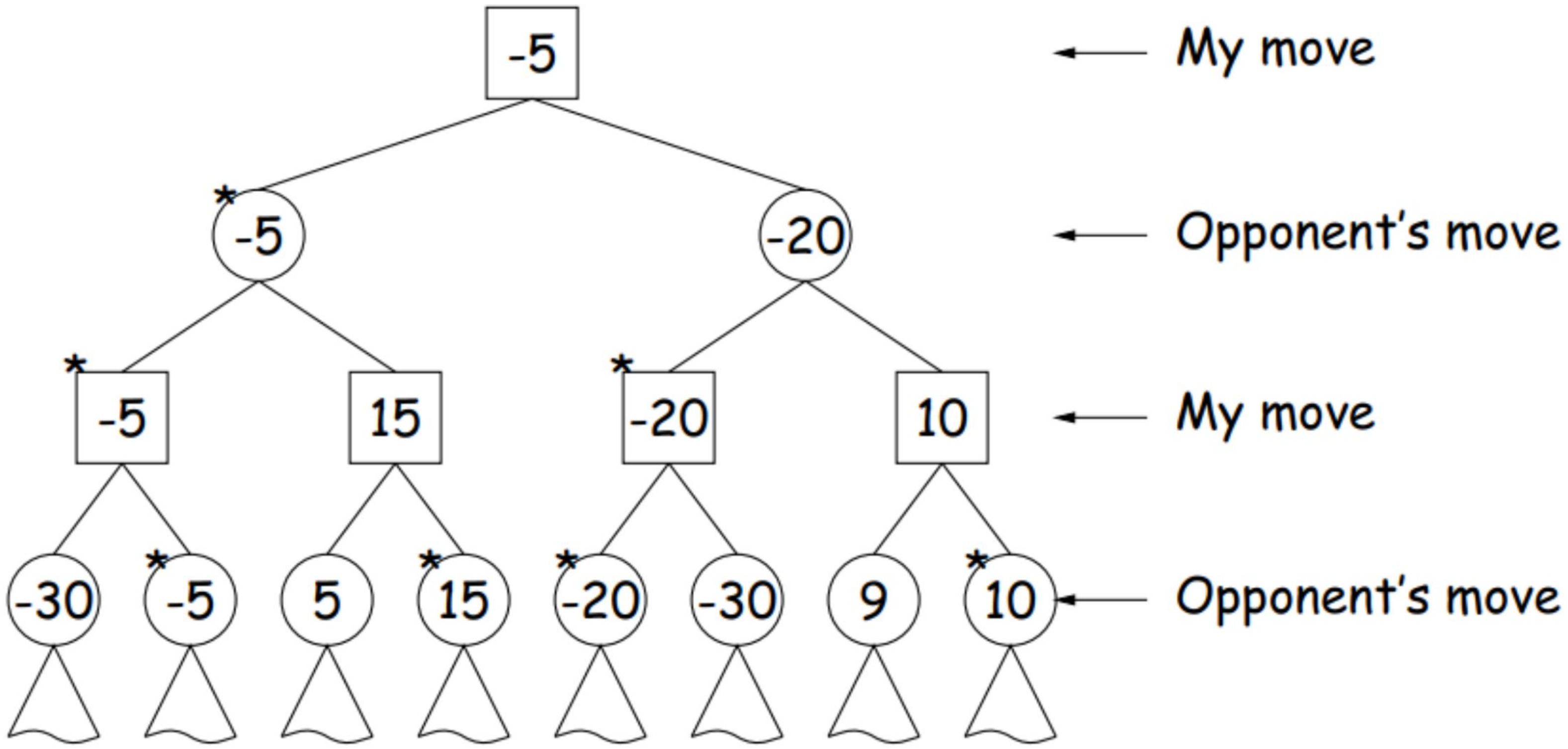
- In tic-tac-toe, we can easily see to the end of the game
- In chess, there are 20~35 legal moves each turn
  - Hence an exponential growth of states we must consider as we search
- So we don't search to the game's end, but instead stop after a certain number of moves (depth)
- We evaluate the board 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 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
```

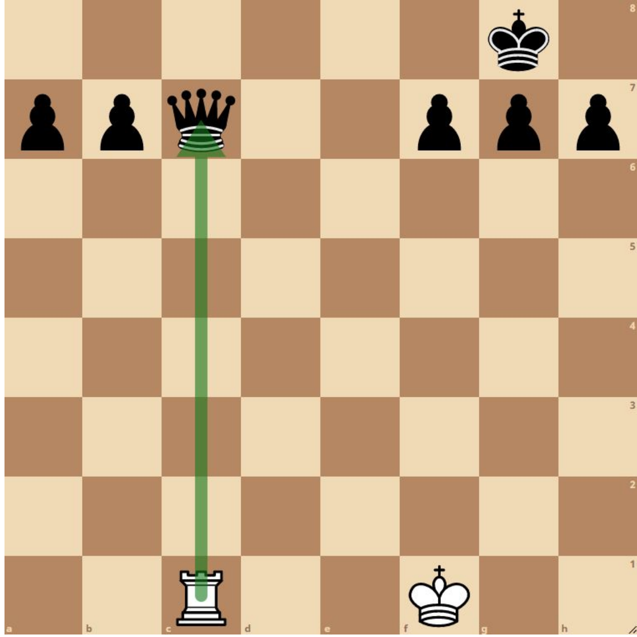
```
findMin(position, depth):
    if (depth == 0 || gameOver(position)):
        return heuristic(position)

    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
```

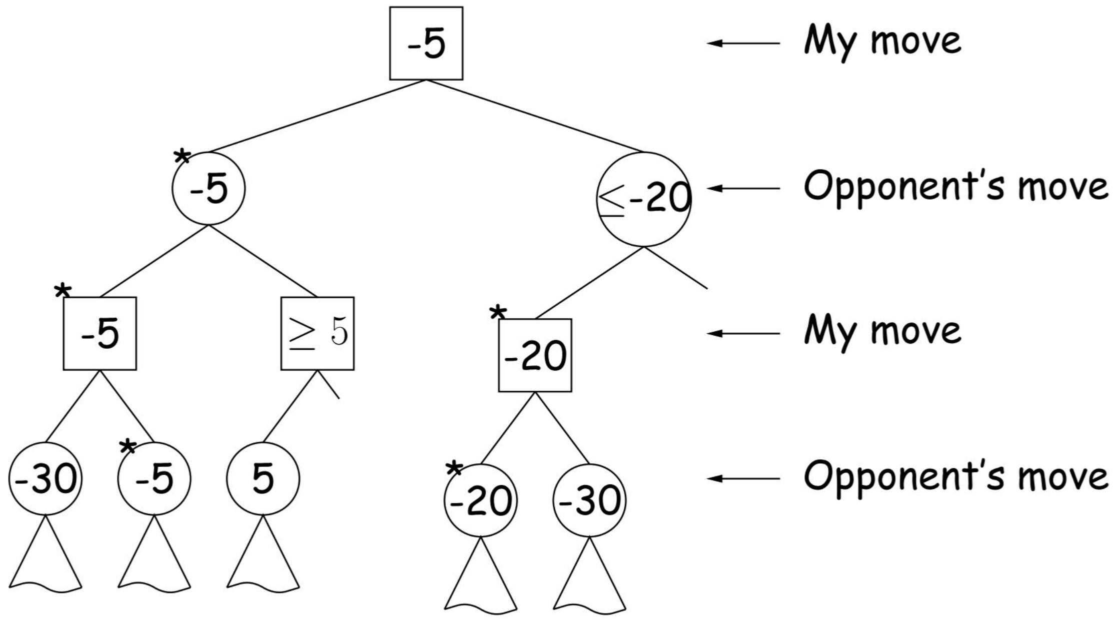
“Sister” functions which call each other: one for min, one for max

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

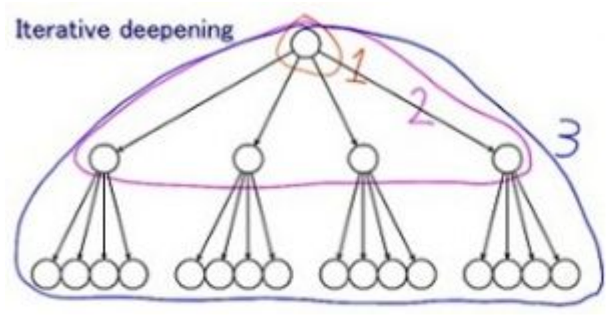
beta = the best (highest) result the min player would accept

What About Time?

- Currently, our search is run with depth as an estimate of “how much searching we do”
- In games like chess, there are limitations on the amount of time used by each player
  - Selecting depth is not a realistic 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



- A depth of search 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