

Game AI

Eric Clinch, Kyle Chin, Sarah Wenger

Intro

- Perfect Information / Imperfect Information Games
- Zero Sum / Non-zero Sum Games
- Game States & Moves
- Game Tree
- Branching factor

Zero-Sum Games

0

Σ



Perfect and Imperfect Information

- In perfect information games, all players know the exact state of the game
 - Chess, Checkers, Go, Monopoly



- In imperfect information games, there is some hidden information that not all of the players have
 - Poker, Hearthstone, StarCraft, League of Legends

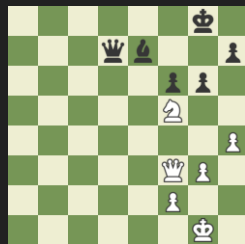


Zero-Sum Games

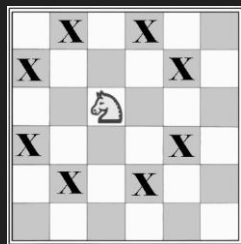
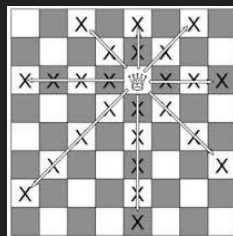
- One player wins and everyone else loses. A good move for me is a bad move for my opponent!
 - Chess
 - Tic-tac-toe
 - Monopoly
 - (most) real life sports
- Non-zero sum games have moves that can benefit both players!
 - Trade/negotiation
 - Studying/collaboration

Game States, Game Moves

- Games have states, i.e., what the current “board” looks like.

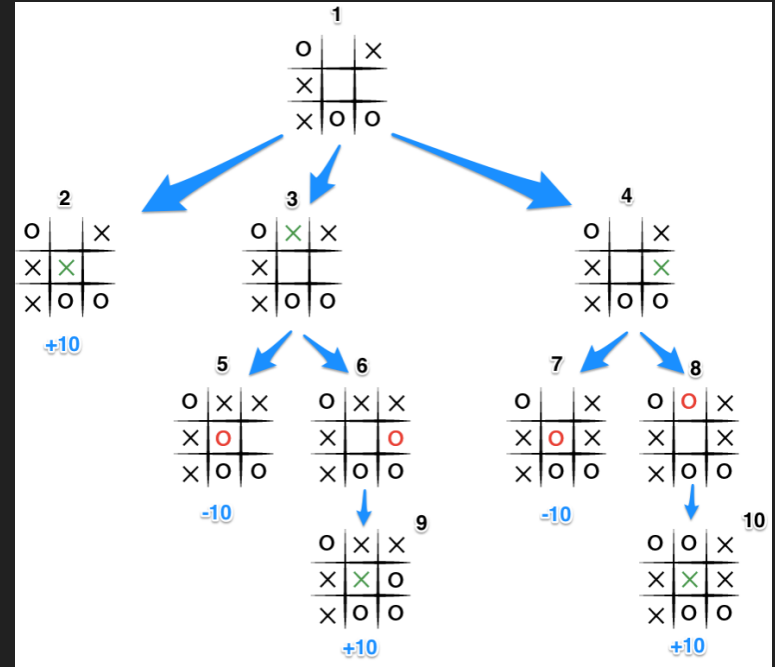


- Games have moves, i.e., what you can do from each board
 - Chess has a lot of possible moves from each board - each piece can move in various ways



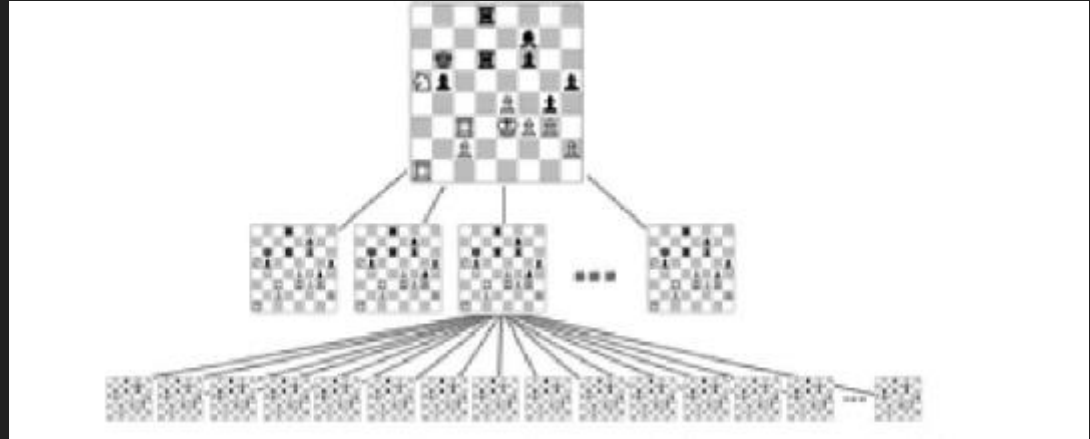
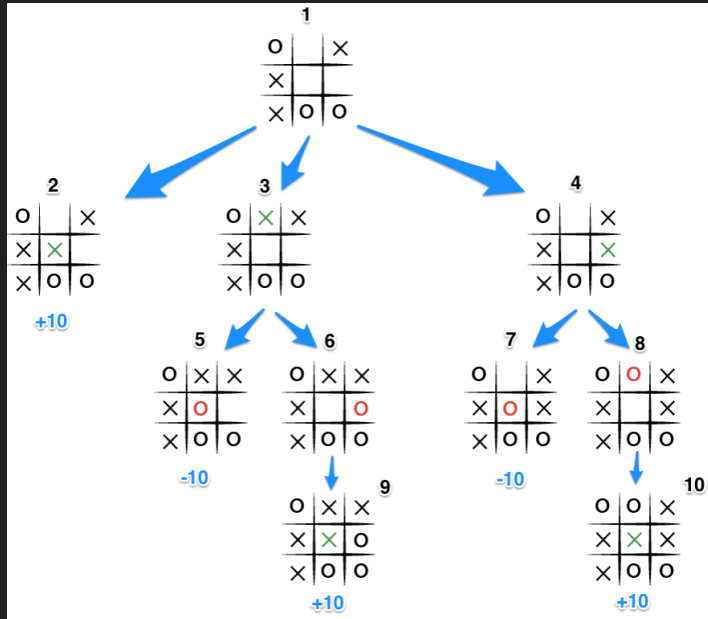
Game Tree

Represents the possible ways the game can play out from the starting state, with game states being the nodes of the tree and moves being the edges in the tree



Branching Factor

Represents how quickly the game tree grows as you go down it. Is approximately the number of moves you can make at each game state.



2-Player Zero-Sum Games

- 2 players, Minnie and Maxie
- If either player wins the game, they get a score of ∞
- Zero-Sum: Minnie score = - Maxie score
- Moves alternate between Maxie's turn and Minnie's turn
- For convenience later on, we will say that the “board score” is the score of Maxie
- Maxie tries to maximize the board score to maximize his own score;
Minnie tries to minimize the board score to maximize her score

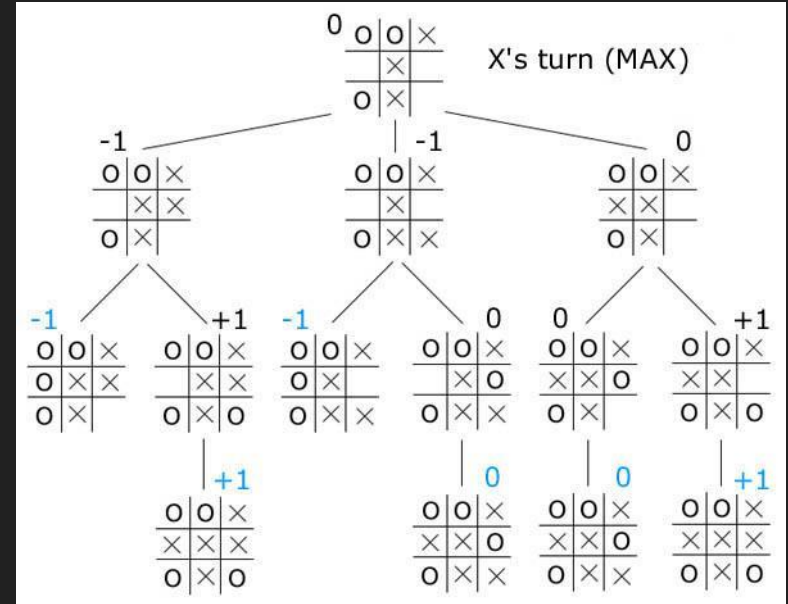


Minimax Algorithm



Minimax Algorithm

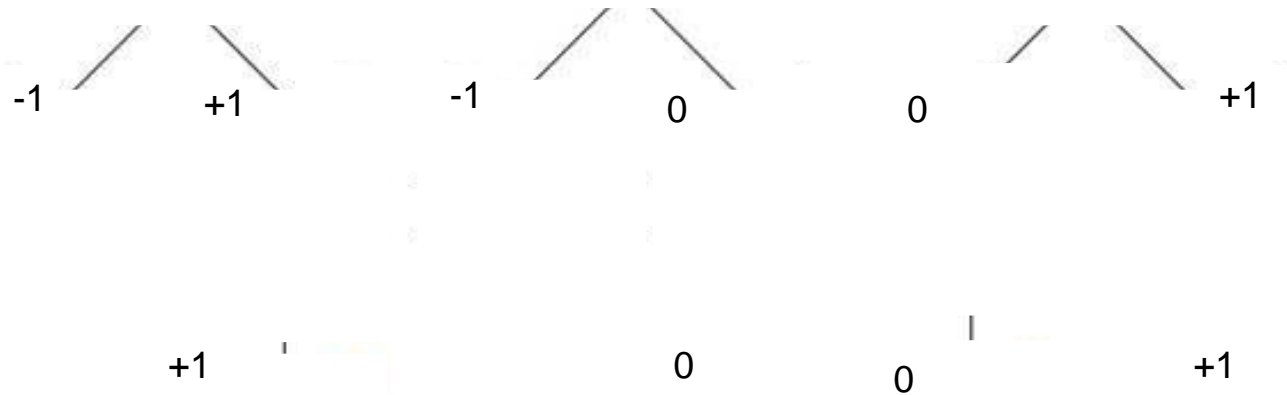
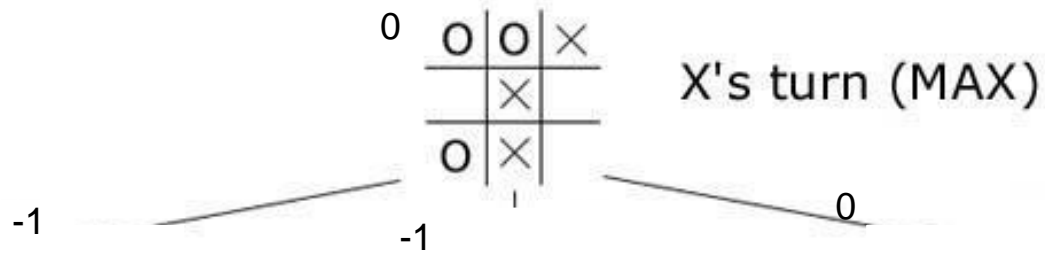
- Motivation: when deciding whether we should make a move, we should consider:
 - What moves our opponents can use to best counter it.
 - And then what moves we could use to best counter that.
 - ...And so on and on



Minimax Algorithm

Algorithm for deciding
what move to make

```
1 # takes a board and returns a tuple (move, score) where move is the
2 # best move for Maxie and score is the board score that results
3 # from making that move. The best move is the one that maximizes
4 # Maxie's score by maximizing the board score
5 def MaxieMove(board):
6     if board.gameOver():
7         return (None,  $\infty$ ) if board.won(Maxie) else (None,  $-\infty$ )
8     else:
9         bestMove = None
10        bestScore =  $-\infty$ 
11        for move in board.legalMoves(Maxie):
12            board.makeMove(move)
13            _, moveScore = MinnieMove(board)
14            board.undoMove(move)
15            if moveScore > bestScore:
16                bestScore = moveScore
17                bestMove = move
18        return (bestMove, bestScore)
```



Minimax: Depth & Heuristics

- In reality, most game trees are too big, so we can't realistically search the entire game tree in this way
- Solution: add a depth parameter
 - When tree reaches a certain depth, stop and use a heuristic to generate a score for the state
- Heuristic function: takes a state, and returns a score based on who is “winning” the current state
 - Higher values (up to ∞) → Maxie is winning
 - Lower values (down to $-\infty$) → Minnie is winning
- A better heuristic function makes your minimax better!

Minimax: Depth & Heuristics

Assume we have some max depth that we stop searching at, and a heuristic function that returns an approximate board score for the given board

```
# takes a board and depth and returns a tuple (move, score) where move is the
# best move for Maxie and score is the board score that results
# from making that move. The best move is the one that maximizes
# Maxie's score by maximizing the board score.
# If depth is the max depth, returns the score given by a heuristic function
def MaxieMoveWithHeuristics(board, depth):
    if board.gameOver():
        return (None, float('inf')) if board.won(Maxie) else (None, float('-inf'))
    else if depth == maxDepth:
        return (None, heuristic(board))
    else:
        bestMove = None
        bestScore = float('-inf')
        for move in board.legalMoves(Maxie):
            board.makeMove(move)
            _, moveScore = MinnieMoveWithHeuristics(board, depth + 1)
            board.undoMove(move)
            if moveScore > bestScore:
                bestScore = moveScore
                bestMove = move
        return (bestMove, bestScore)
```

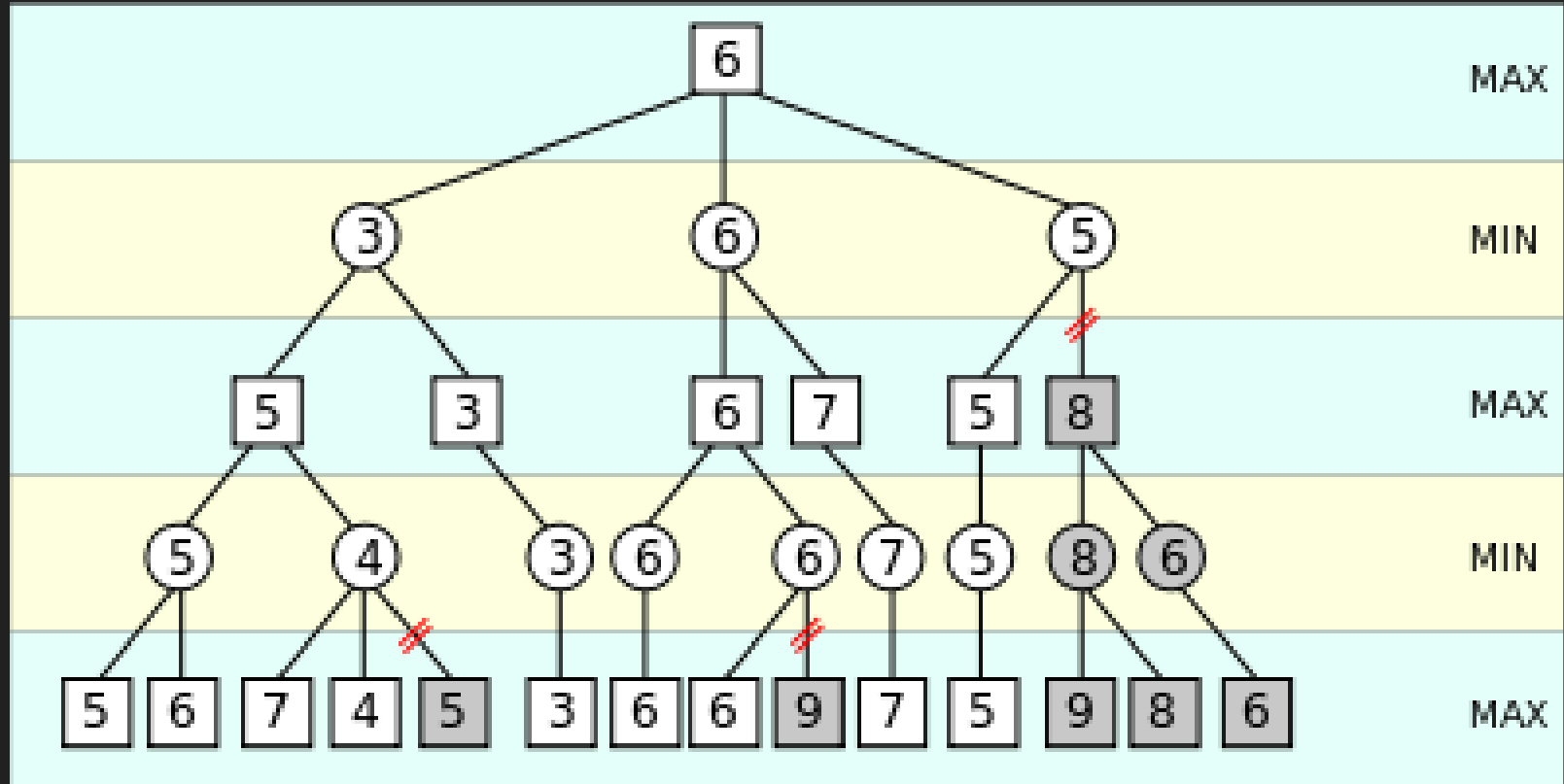


AUTO✓LAB

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |



Alpha-Beta Pruning



Alpha-Beta Pruning

- Augmentation of minimax to reduce branches of the game tree
- Motivation: if we know that a branch of the game tree is too good to be true (ie, our opponent would never let us get to this branch), then don't waste time considering it, because we probably won't end up in that branch anyway
- Cuts off branches based on two extra parameters: Alpha and Beta
 - Alpha: best guaranteed score of Maxie
 - Beta: best guaranteed score of Minnie
- If $\beta \leq \alpha$, then this branch is “too good to be true” and we don't have to continue down that branch of the game tree.

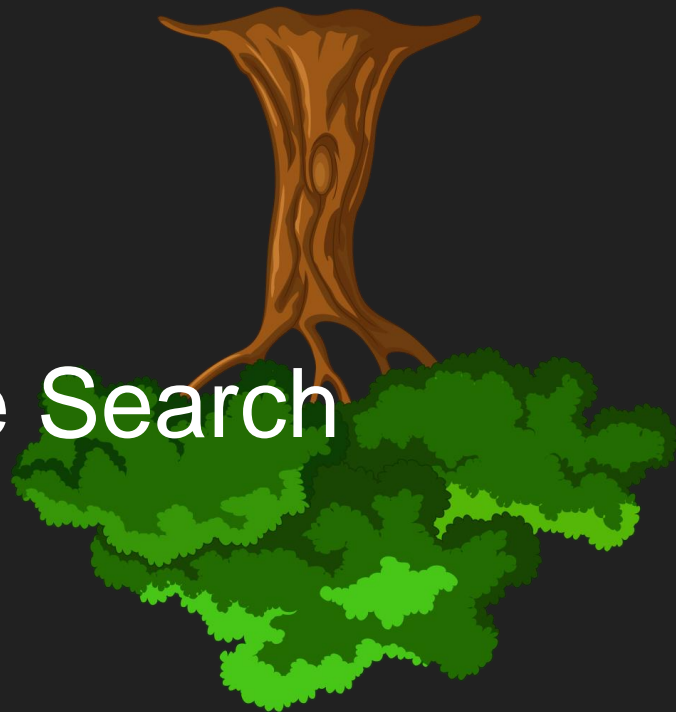
Alpha-Beta Pruning

```
1 # takes a board and returns a tuple (move, score) where move is the
2 # best move for Maxie and score is the board score that results
3 # from making that move. The best move is the one that maximizes
4 # Maxie's score by maximizing the board score
5 def MaxieMove(board, depth, alpha, beta):
6     assert(alpha < beta)
7     if board.gameOver():
8         return (None,  $\infty$ ) if board.won(Maxie) else (None,  $-\infty$ )
9     else if depth == maxDepth:
10         return (None, heuristic(board))
11     else:
12         bestMove = None
13         bestScore =  $-\infty$ 
14         for move in board.legalMoves(Maxie):
15             board.makeMove(move)
16             _, moveScore = MinnieMove(board + 1, alpha, beta)
17             board.undoMove(move)
18             if moveScore > bestScore:
19                 bestScore = moveScore
20                 bestMove = move
21                 alpha = max(alpha, bestScore)
22                 if (alpha >= beta):
23                     return (bestMove, bestScore)
24         return (bestMove, bestScore)
```

Minimax with Alpha-Beta Pruning

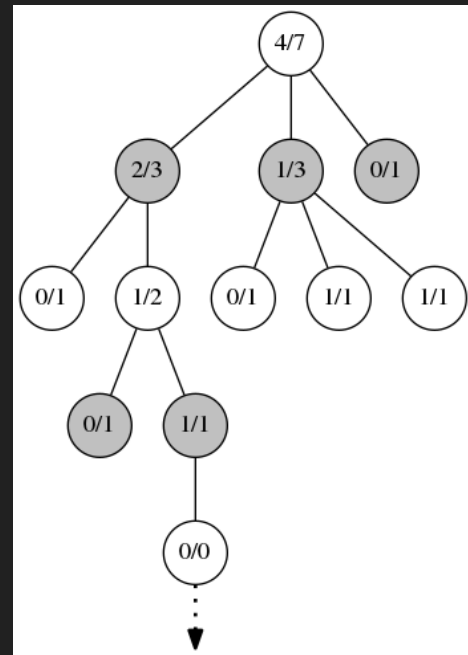
- Minimax with alpha-beta pruning is very good at playing 2-player, zero-sum, perfect information games with relatively small branching factors.
 - Great for games like chess, checkers, othello
 - Deep Blue, the first chess programmer that was able to play at a superhuman level, used the minimax algorithm with alpha-beta pruning
- Unfortunately, it can't be extended to non-zero sum games or games with imperfect information
- Minimax also tends to perform poorly on games with large branching factors as it spends too much time evaluating moves that are ultimately useless
 - Bad at games like Go, real time strategy games
 - Idea: could we try to find out which moves are more promising early on and focus on those moves, rather than focusing equally on every move?

Monte Carlo Tree Search



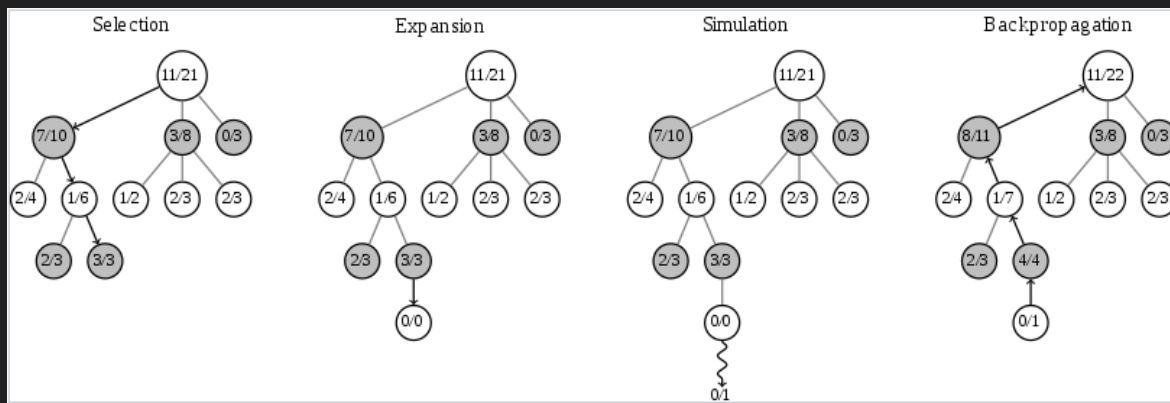
Monte Carlo Tree Search

- Rather than try searching the entire game tree up to some depth like minimax does, we will construct the game tree by sampling the game tree a little bit at a time, and over time we will focus on sampling from moves that are more promising.
 - At each node of the game tree we construct, we will store the number of times that node has been sampled, as well as the node's score (how good that node is).

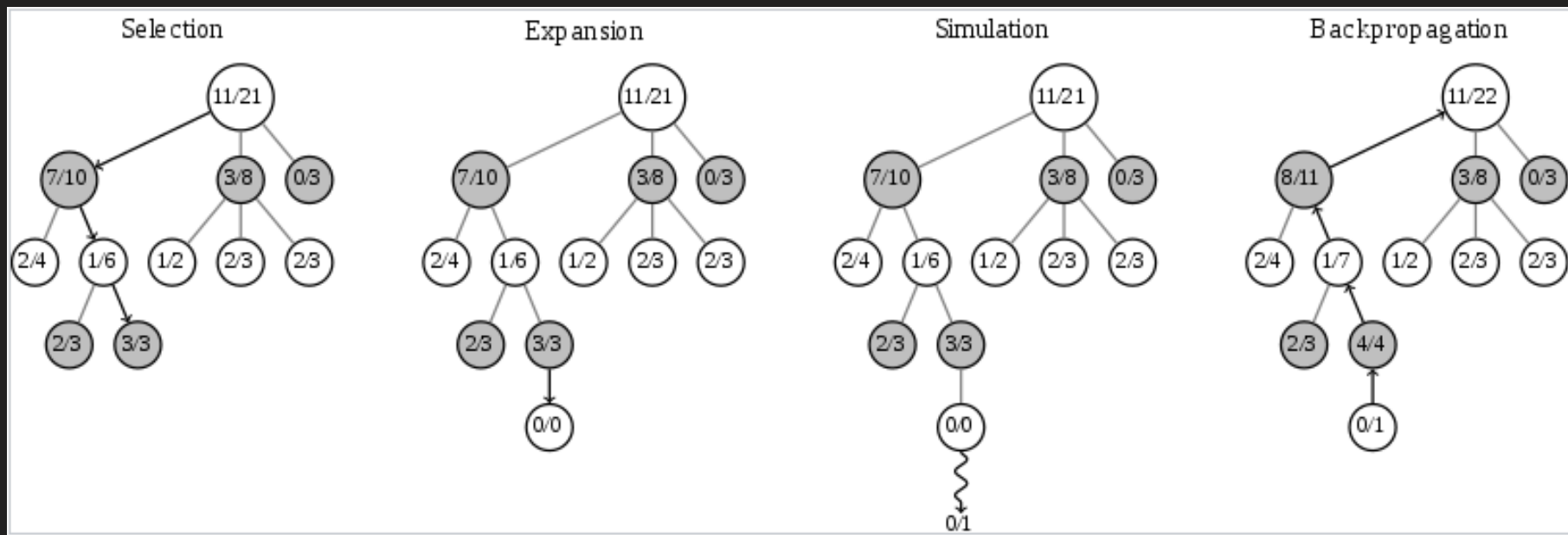


Monte Carlo Tree Search

- Each iteration of the algorithm consists of 4 parts:
 - Selection: select which part of the game tree to sample, starting from the top and working your way down to a leaf (bottom) node
 - Expansion: pick one of the unsampled moves from leaf node and add it to our game tree
 - Simulation (random playout): as a heuristic, randomly pick moves for each player until the game is over
 - Backpropagation: propagate the result of the simulation up the game tree, updating the number of samples and score of each node on the branch on the way up



Monte Carlo Tree Search



Exploration vs Exploitation

- In the selection phase, how do we decide which move to pick?
- The exploitation vs. exploration problem: should we always select the node with the highest average score so far (exploitation) in order to focus on the most promising move, or should we pick the node that has been sampled the least amount of times (exploration) in order to try to get a more accurate idea of how good that node is?
 - Problem with exploitation: might end up discounting good nodes that just got unlucky on their first few samples
 - Problem with exploration: will focus equally on each move, which defeats the purpose of MCTS

¿Por que no los dos? (Why not both?)

- Pick the node i that maximizes the following function:

$$\frac{w_i}{s_i} + c \sqrt{\frac{\ln s_p}{s_i}}$$

- Where w_i is the number of wins for node i , s_i is the number of samples for node i , s_p is the number of samples of the parents node, and c is some constant (theoretically $\sqrt{2}$, but generally chosen empirically)

UCB1 algorithm

$$\frac{w_i}{s_i} + c \sqrt{\frac{\ln s_p}{s_i}}$$

- The first term represents exploitation, as it is maximized by nodes with higher average scores
- The second term represents exploration, as it is maximized by nodes that have been visited less

Monte Carlo Tree Search

- Because it focuses most of its time on more promising moves, it can scale well to games with high branching factors like Go and real-time strategy games.
- Because it doesn't require a specialized heuristic and just uses random playouts, it can be applied to nearly any 2-player, zero-sum, perfect-information game and achieve good results.
- Unlike the minimax algorithm, MCTS can be modified to play imperfect-information games.

Attendance: Thanks for Coming! :)

Other cool algorithms

- For pathfinding/shortest path: Dijkstra's Algorithm and the A* algorithm
- To achieve some goal (solving a Rubik's cube, escaping from a maze) in the shortest number of moves: Breadth First Search