Outline

- Review history of AI’s success in game play
- Describe the game of Go
- Introduce Monte Carlo Tree Search
- Discuss advantages and disadvantages of MCTS

Checkers

- Average game is 50 moves long with a branching factor of 6
- Arthur Samuels worked at IBM in 1949
- Built a checkers playing program on a computer with no operating system
- His program learned how to weight features of his static evaluation function through self play
- Achieved the level of a human player
- Samuels was the first to coin the term machine learning

Chess

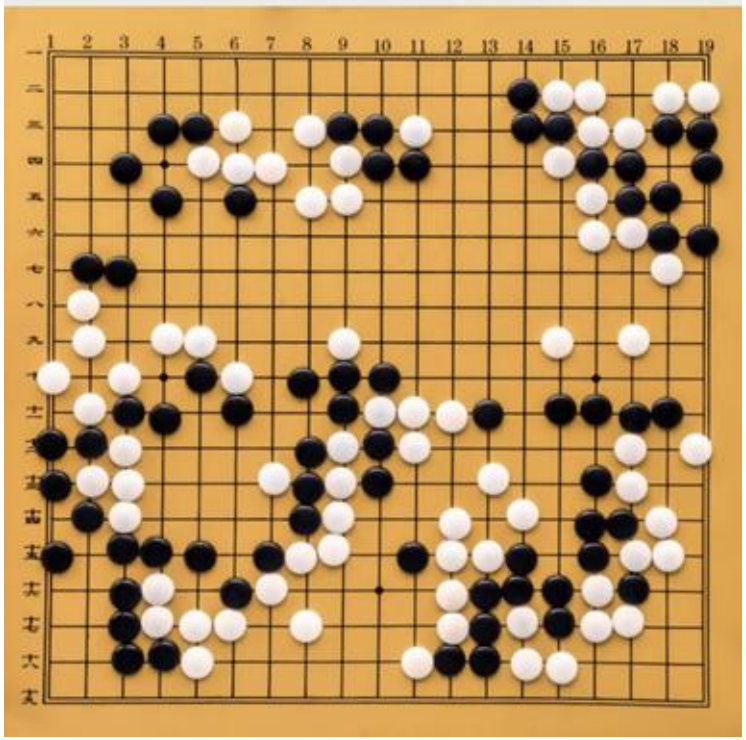
- Average game is 40 moves long with a branching factor of 35
- IBM researchers created a chess playing program named Deep Blue
- Used no machine learning, just a deeper search helped by a specialized parallelized architecture and an elaborate static evaluation function
- Deep Blue defeated the human world champion Kasparov in 1997
- Deep Blue is a chess savant, but has no general intelligence

Claude Shannon said that a machine that can surpass a human at chess “will force us to admit the possibility of mechanized thinking or to further restrict our concept of thinking.”

John McCarthy said that “as soon as it works, no one calls it AI anymore.”

Go

- Played on a 19 x 19 board
- Each of the 361 points (intersections of lines on the grid) are a possible place to play
- Players alternate turns either placing a white stone or black stone
- When stones are surrounded (with no empty points) they are captured and removed
- The goal is to control the most points



Why Go is so challenging for AI

- The large branching factor makes it difficult to apply Minimax
  - 361 possible first moves, 360 possible second moves, and so on
  - Average branching factor is 250
  - Impossible to search deeply
- It is also difficult to create a good static evaluator
  - Unlike in Chess, each piece is of equal value
  - The goodness of a board state depends on a complex analysis of which groups of stones could form a connected groups

AlphaGo vs Lee Sedol

- Many believed humans have an advantage over machines at Go
- The ability to recognize visual patterns is essential
- AlphaGo defeated the human world champion Lee Sedol in 2016
- AlphaGo learned through self-play using
  - Reinforcement learning
  - Convolutional neural networks
  - Monte Carlo Tree Search
- We will learn MCTS this today!

Monte Carlo Tree Search

- A method for making game play decisions
- Combines randomness with tree search
- Can be applied to any domain that can be described in terms of states and actions
- Recent interest in this method is due to its major successes when applied to Go

Comparing game playing methods

Minimax

- For each move performs a new search
- Searches to a depth bound
- Doesn’t create nodes, uses recursion to do a depth-limited, depth-first search
- Backs up information found at the depth-bound

Monte Carlo Tree Search

- For each move adds information to an existing search tree
- Does simulated rollouts to the end of the game to gain information about which possible move to take
- Each node stores the number of wins and the number of visits

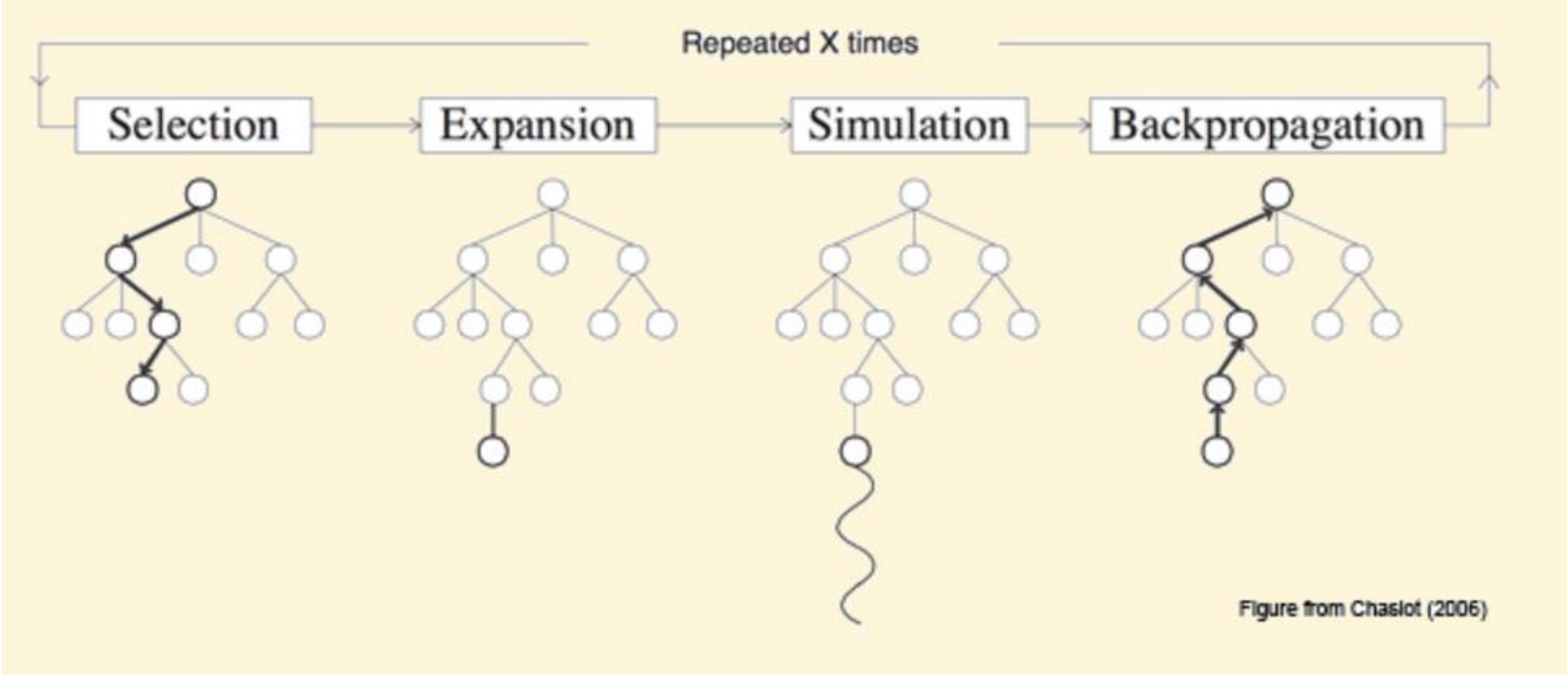
Four phases to the MCTS algorithm

Repeat these phases multiple times before choosing the next move:

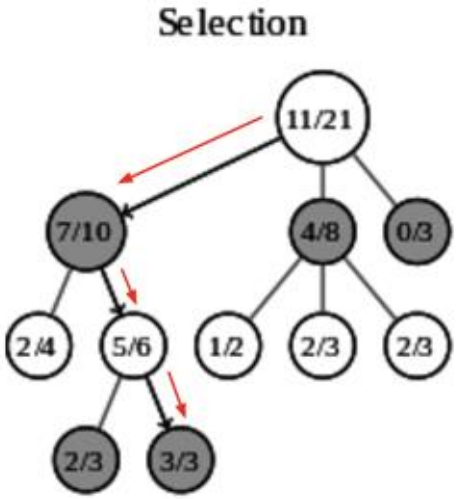
1. **Selection**: Starting at the current node, select child nodes until a node is reached that is not fully expanded
2. **Expansion**: If the selected node is not terminal, choose one of its children
3. **Simulation**: Run a simulated roll out from that child until the end of the game is reached
4. **Backpropagation**: Update all nodes along the path of selection to expansion with the simulation results

Use the accumulated statistics in the tree to choose a move

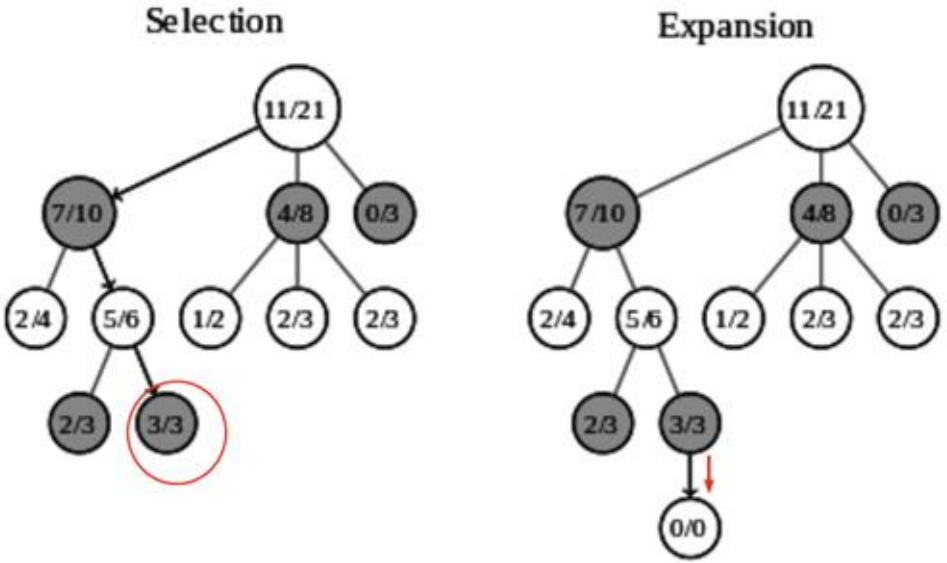
Visualizing MCTS



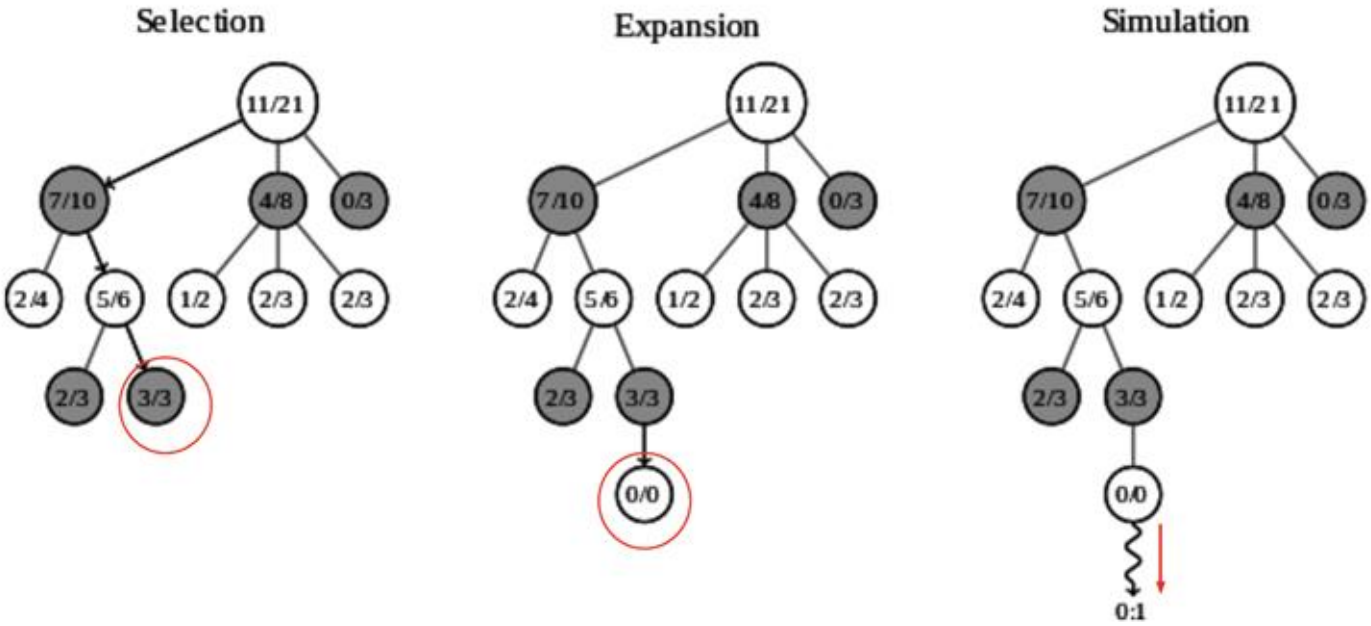
Selection: White's turn, select "best" nodes until a node is reached that is not fully expanded



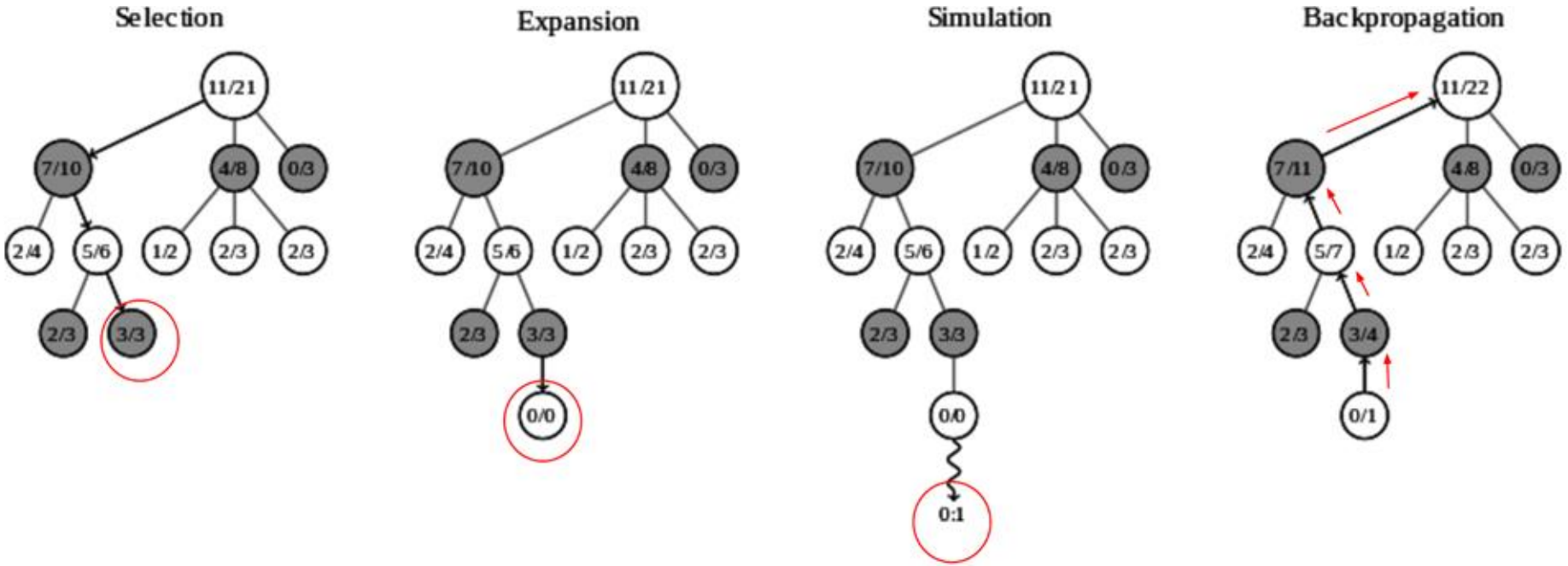
Expansion: If node is not terminal, choose one of its children to expand



Simulation: Run a simulated roll out until the end of the game is reached (White lost)



Backpropagation: Update nodes on path to expansion with rollout results



How to determine “best” nodes to select

Want to maintain a balance between

- **Exploiting** moves with high average winning rate
- **Exploring** moves with few visits
- Choose a node that maximizes a reward function

Maximize the Upper Confidence Bound (UCB)

$$v_i + C * \text{sqrt}(\ln(N) / n_i)$$

- v_i is the estimated value of the node based on rollouts
  - Higher values **increase** selection, which encourages **exploitation**
- n_i is the number of times the node has been visited
  - Higher visits **decrease** selection, which encourages **exploration**
- **N** is the number of times the parent has been visited
- **C** is a tunable bias parameter

Representing the value of a node (v_i in UCB formula)

- Possible outcomes of a game are **win**, loss, or draw
- Value should always be positive
- Should reflect the current player's chances of winning
  - When outcome is -1, this is a loss for player1 and a win for player2
  - When outcome is +1, this is a win for player1 and a loss for player2
- Store all values from player1's perspective, we will adjust them for player2

Representing the value of a node (v_i in UCB formula)

$$v_i = 1 + ((wins-losses)/gamesPlayed)$$

What would the value be if the agent:

- Played 10 games and won them all?
- Played 10 games and lost them all?
- Played 10 games and drew them all?
- Played 10 games and won 7 and lost 3?
- Played 10 games and won 3 and lost 7?
- What are the range of possible values?

Representing the value of a node (v_i in UCB formula)

$$v_i = 1 + ((wins - losses)/gamesPlayed)$$

- Adding 1 ensures that the quantity will be positive
- Maximum value is 2, when all outcomes are wins
- Minimum value is 0, when all outcomes are losses
- A player that draws more than loses will have a higher value
- Just as in Minimax, we must swap perspectives for each player
  - MAX nodes use v_i directly
  - MIN nodes use $2 - v_i$

Goal: Maximize UCB

$v_i + C * \text{sqrt}(\ln(N) / n_i)$

Assume

- **C = 1**
- **v_i = 0.5**

As parent visits increase while node visits remain fixed, the more likely we are to choose this node

N	n_i	UCB value
10	10	1.18
20	10	1.27
30	10	1.32
40	10	1.36
50	10	1.38
60	10	1.40
70	10	1.42
80	10	1.44
90	10	1.45
100	10	1.46

Goal: Maximize UCB

$v_i + C * sqrt(ln(N) / n_i)$

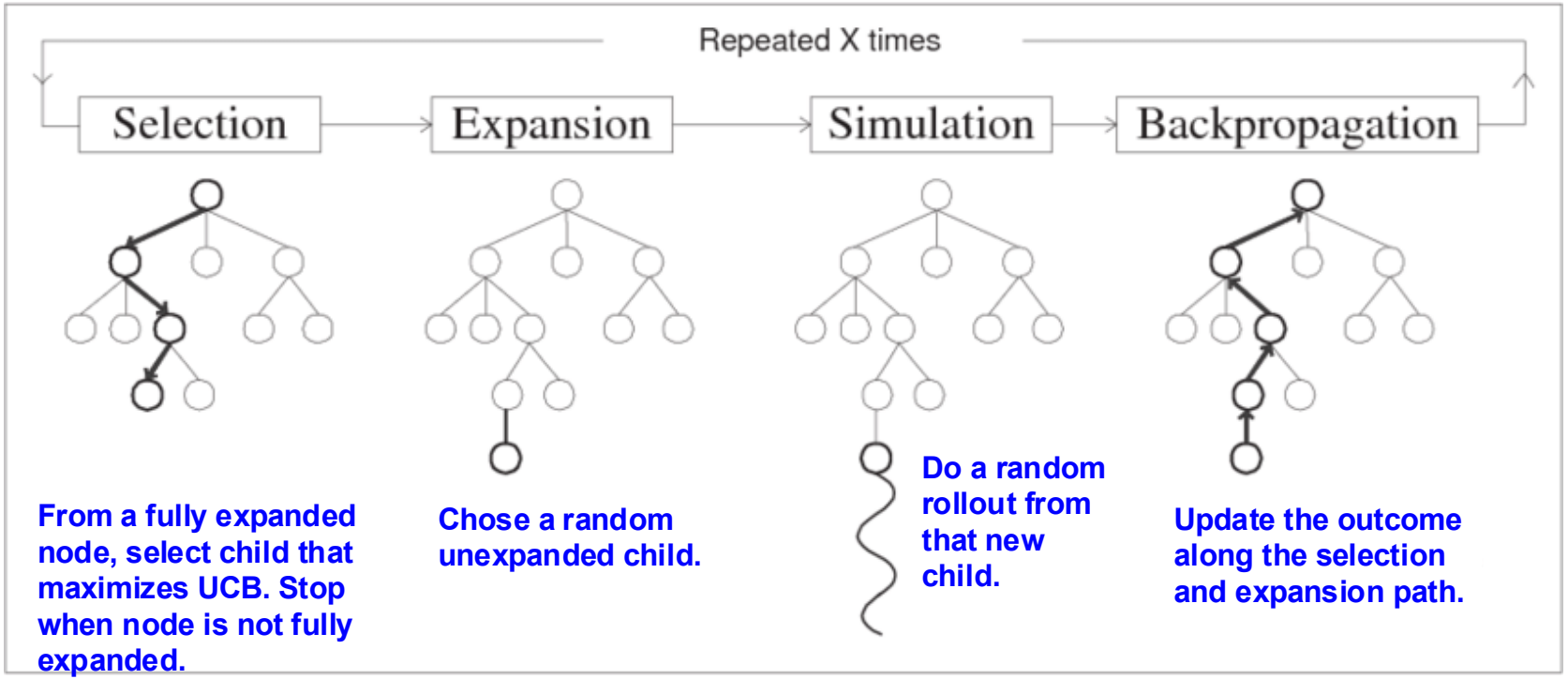
Assume

- **C = 1**
- **v_i = 0.5**

As node visits increase (so too will parent visits), the less likely we are to choose this node

N	**n_i**	**UCB value**
10	5	1.18
20	15	0.95
30	25	0.87
40	35	0.82
50	45	0.79
60	55	0.77
70	65	0.76
80	75	0.74
90	85	0.73
100	95	0.72

MCTS: Interface for each phase of the algorithm



Overview of MCTS

Repeat `num_rollouts` times before choosing next move:

1. Selection: Starting at the current node, as long as the current node is fully expanded, choose successor with maximum UCB
2. Expansion: Randomly choose one of the unexpanded successors to add to the tree
3. Simulation: Run a simulated rollout from there
4. Backpropagation: Update all nodes on the selection/expansion path with the results

Each rollout adds one node to the tree (do as many as possible)

When each phase of MCTS is used

Selection

- Used for nodes we've seen before that are fully expanded
- Picked according to UCB

Expansion

- Used when we reach the frontier
- Add one child per rollout

Simulation

- Used beyond the search frontier
- Pick moves randomly

Backpropagation

- Used after reaching a terminal state
- Update value and visits of all nodes in selection/expansion path

Advantages of MCTS

1. Does not require any domain knowledge
    - Simply needs to know the legal moves and when the game ends
    - A single implementation can be used for many different games
1. Focuses on the most relevant branches of the game tree
    - Grows tree asymmetrically
    - Visits more interesting nodes more often
    - Makes it more suitable for games with large branching factors like Go
1. Any time algorithm
    - Can be halted at any time and will return best move found so far

Disadvantages of MCTS

1. Strength of play
    - Can fail to find good moves in a reasonable amount of time
1. Search speed
    - Nodes must be visited many times before they begin to yield reliable estimates