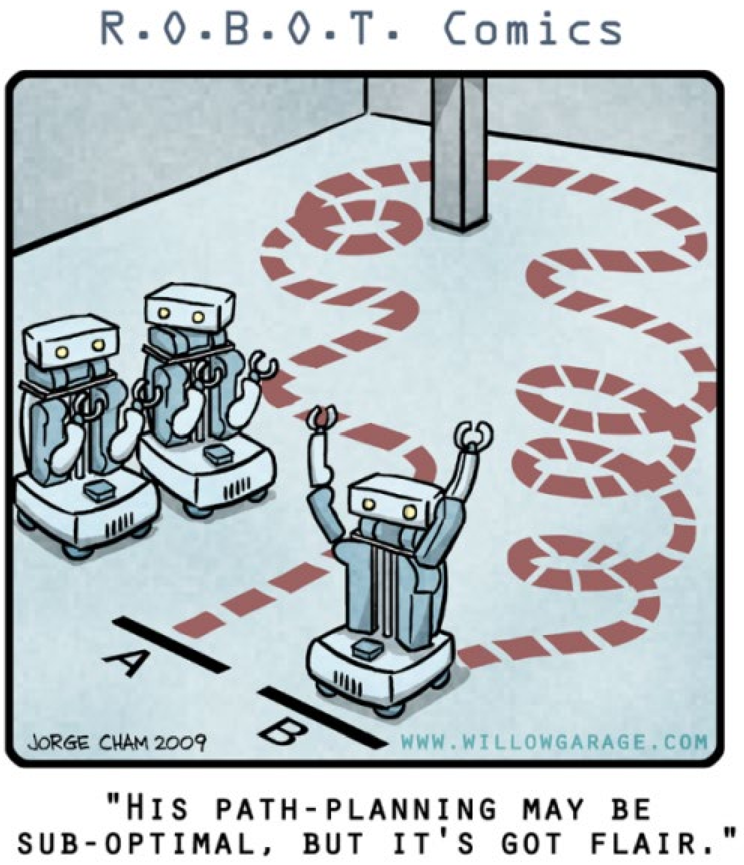


Robot Intelligence: Planning

CS 4649/7649 – Asst. Prof. Matthew Gombolay

Slides adapted from:
MIT 16.410 Emilio Frazolli
CalTech CS 159 Suraj Nair, Peter Kundzicz, Kevin An, Vansh Kumar
Russell and Norvig AIMA

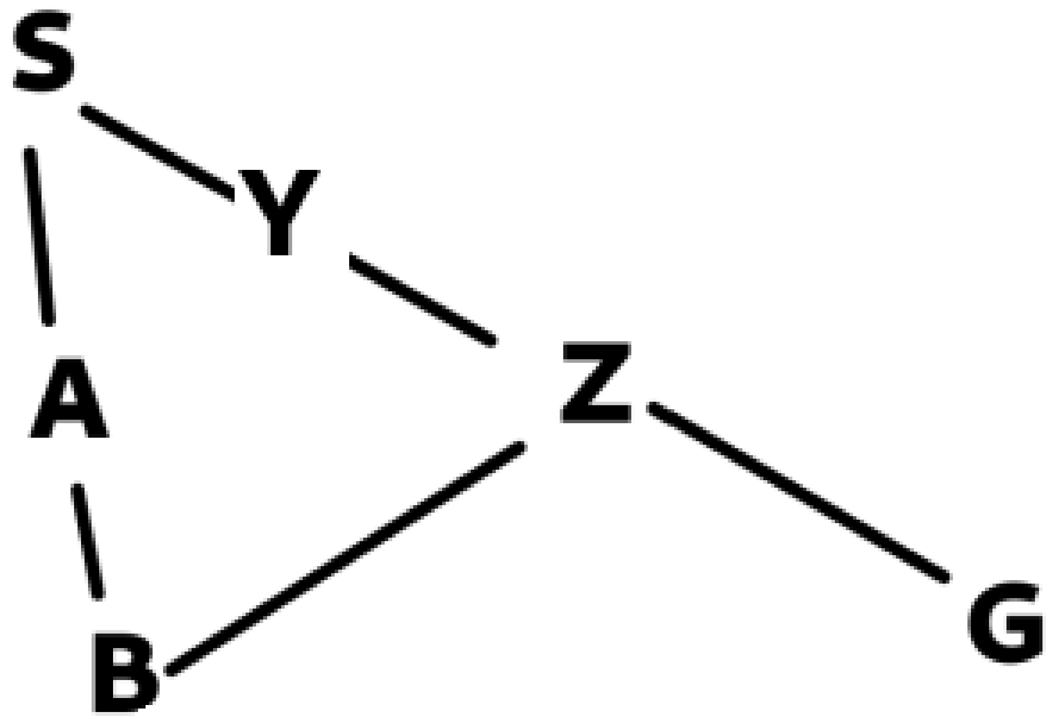
Assignments

- Due Today, 8/26
  - Read Ch. 6 in Russel & Norvig
  - Pset1 due at 1:59 PM Eastern

- Due Monday, 8/31
  - Reading: Ch. 10

- Due Wednesday, 9/02
  - Reading: Ch. 10

IDS



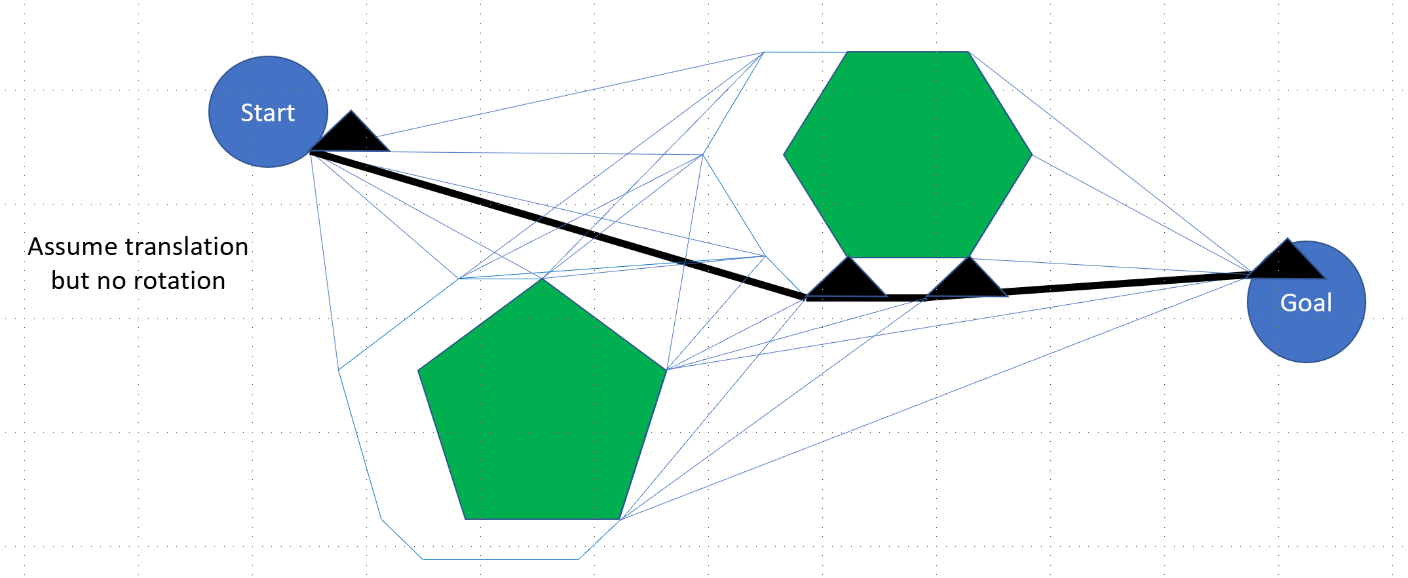
Using a visited list prevents IDS from finding optimal path!

Outline

- Informed search methods: Introduction
  - Shortest Path Problems on Graphs
  - Uniform-cost search
  - Greedy (Best-First) Search
- Optimal Search
- Monte-Carlo Tree Search

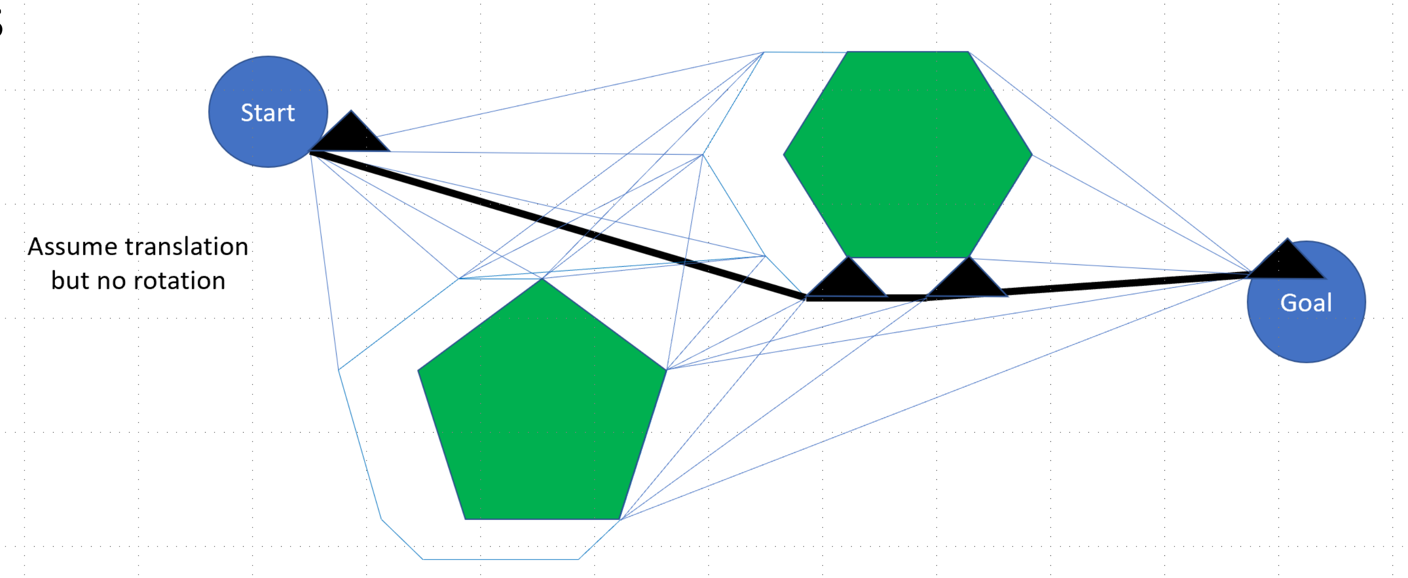
Review

- We can discretize collision-free trajectories into a finite graph.
- Searching for a collision-free path can be converted into a graph search.
- Hence, we can solve such problems using the graph search algorithms discussed in Lectures 2-4(BFS, DFS, etc.).



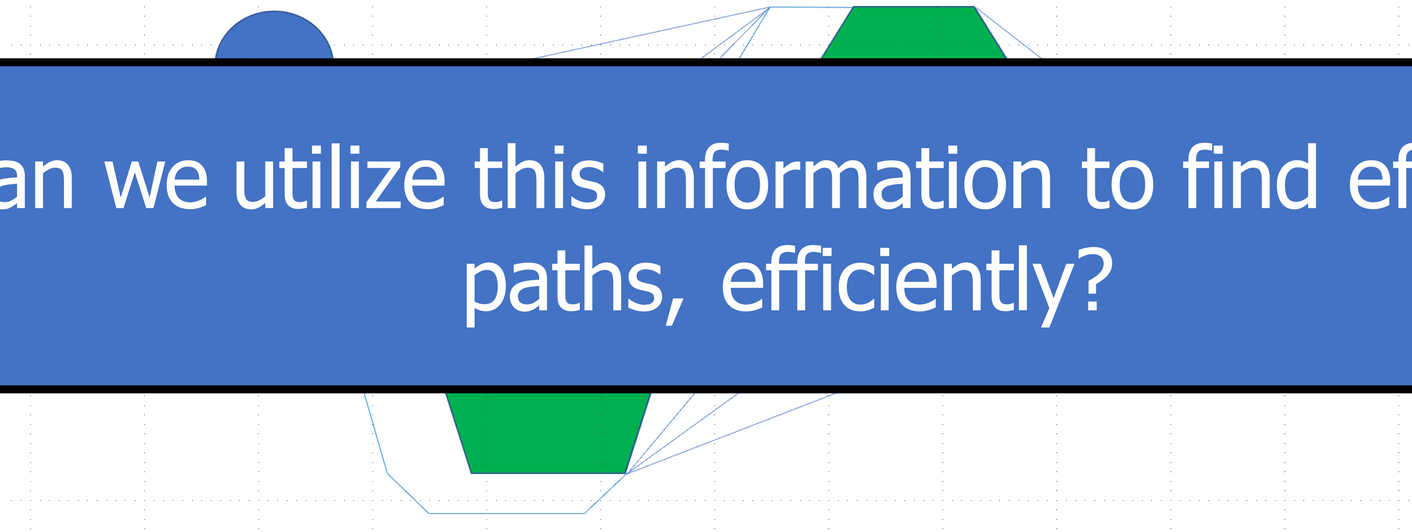
Review

- However, roadmaps are not just "generic" graphs
  - Some paths are much more preferable with respect to others (e.g., shorter, faster, less costly in terms of fuel/tolls/fees, more stealthy, etc.)
  - Distances have a physical meaning
  - Good guesses for distances can be made, even without knowing optimal paths



Review

- However, roadmaps are not just “generic” graphs
  - Some paths are much more preferable with respect to others (e.g., shorter, faster, less costly in terms of fuel/tolls/fees, more stealthy, etc.)
  - Distances have a physical meaning
  - Good guesses for distances can be made, even without knowing optimal paths



Can we utilize this information to find efficient paths, efficiently?

Shortest Path Problems on Graphs

Input: A* Search problem P $= \langle G, v_s, v_g \rangle$ where

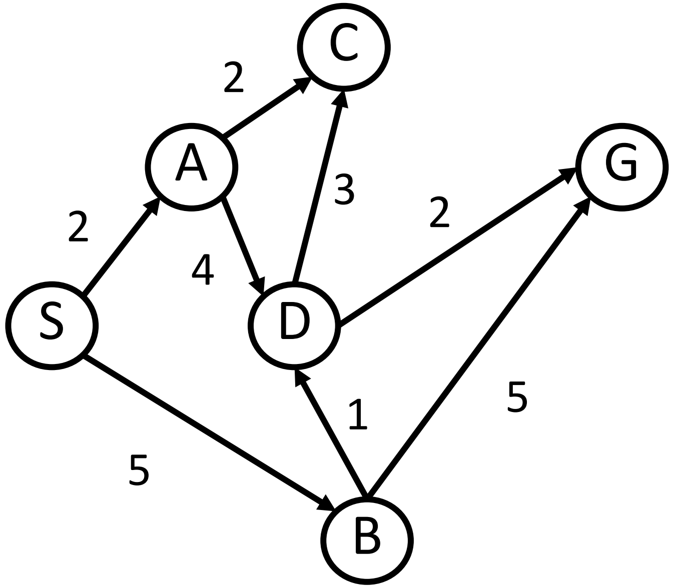
- Graph, $G \;=\; \langle V, E \rangle$
  - V is set of Vertices
  - E is set of Edges
- Start vertex $\mathrm{v_s} \in V$
- Goal vertex $\mathrm{v_g} \in V$
- Cost Function, $w: E \rightarrow \mathbb{R}^+$, that associates each edge to a positive weight (e.g., cost, length, time, fuel)

Output: A simple path, e.g. $P = \langle v_g, \ldots\ , v_s \rangle$, in G from v_s to v_g (i.e., $\langle v_i, v_{i+1} \rangle \in E$ and $v_i \neq v_j$ if $i \neq j$) such that its weight w(P) is minimal among all such paths.

- The weight of a path is the sum of the weights of its edges.

Example: Point-to-point shortest path

Find the minimum-weight path from s to g in the graph below:



Solution: a simple path $P = \langle g, d, a, s \rangle$ with weight $w(P) = 8$.

- ($P = \langle g, d, b, s \rangle$ would be acceptable too)

Uniform-Cost Search

Let Q be a list of partial paths,

S be the start node and

G be the goal node.

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove N from Q
   b) Find all children of head(N) not in Visited and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Uniform-Cost Search

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N   //Goal reach!
4. Else:
   a) Remove partial path N with the lowest cost $w(N)$ from the queue Q
   b) Find all children of head(N) and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

Uniform-Cost Search

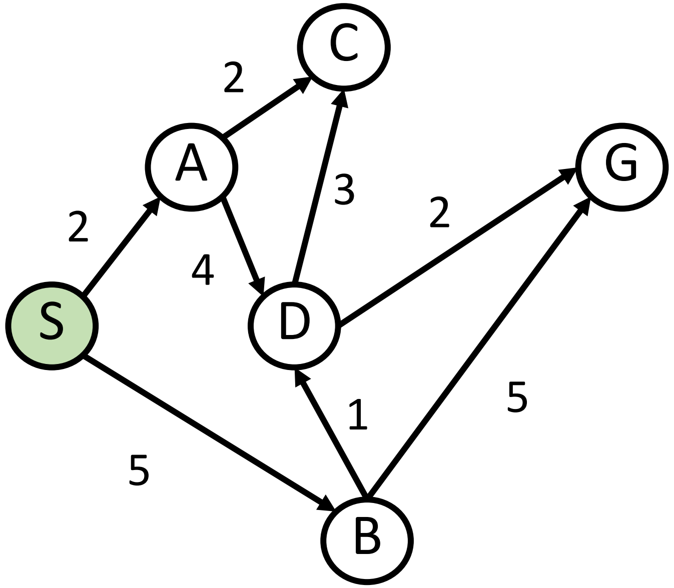
1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N   //Goal reach!
4. Else:
   a) Remove partial path N with the lowest cost $w(N)$ from the queue Q
   b) Find all children of head(N) and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) GOTO step 2

Note: no visited list!

Example of Uniform-Cost Search

Queue

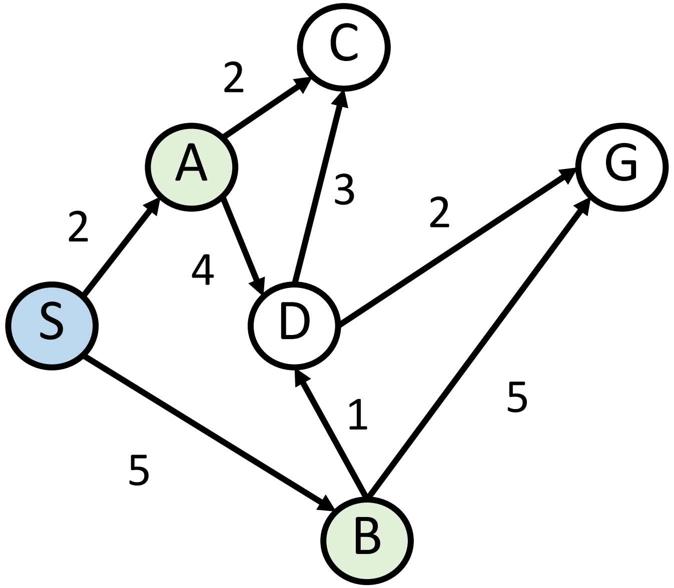
State	Cost
$\langle S \rangle$	0



Example of Uniform-Cost Search

Queue

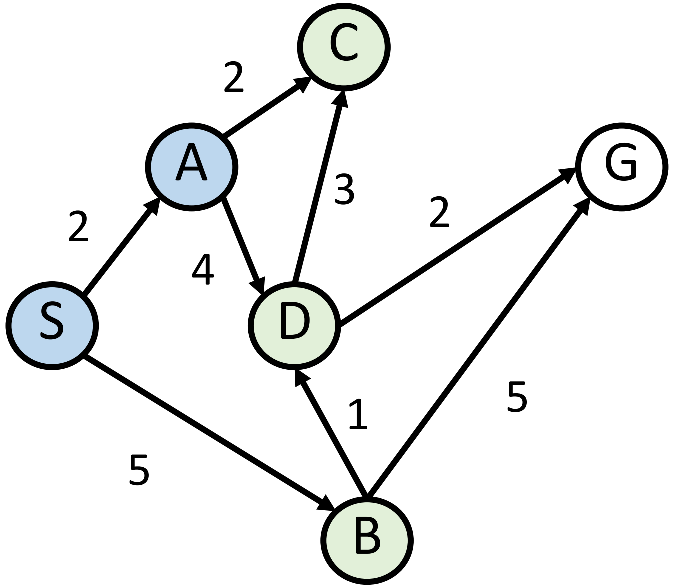
State	Cost
$\langle A, S \rangle$	2
$\langle B, S \rangle$	5



Example of Uniform-Cost Search

Queue

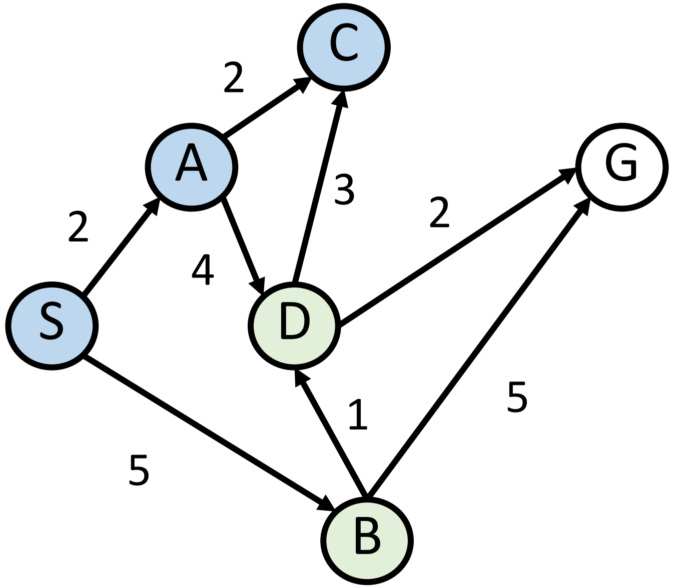
State	Cost
$\langle C, A, S \rangle$	4
$\langle B, S \rangle$	5
$\langle D, A, S \rangle$	6



Example of Uniform-Cost Search

Queue

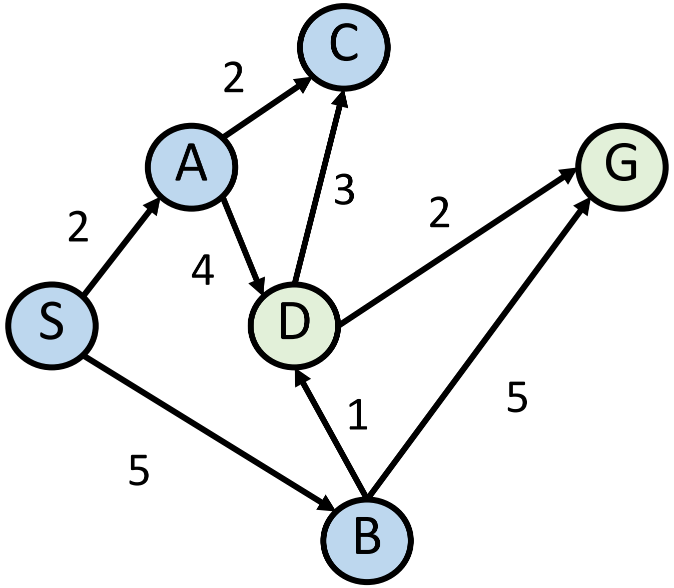
State	Cost
$\langle B, S \rangle$	5
$\langle D, A, S \rangle$	6
$\langle D, C, A, S \rangle$	7



Example of Uniform-Cost Search

Queue

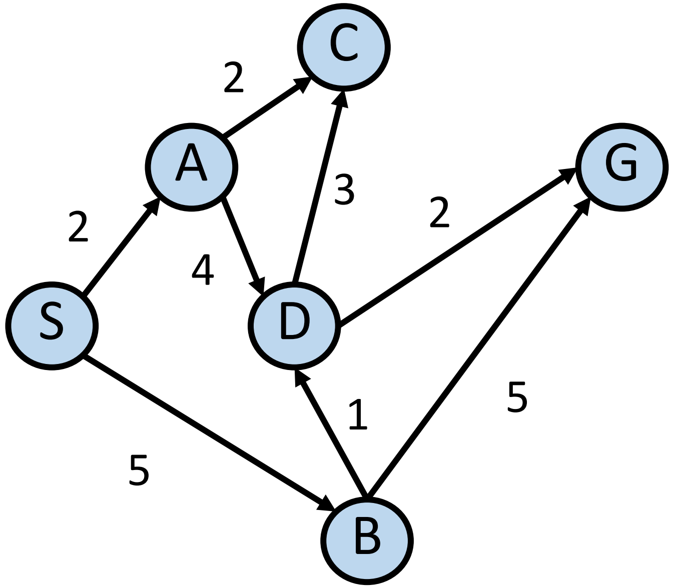
State	Cost
$\langle D, A, S \rangle$	6
$\langle D, C, A, S \rangle$	7
$\langle G, B, S \rangle$	10



Example of Uniform-Cost Search

Queue

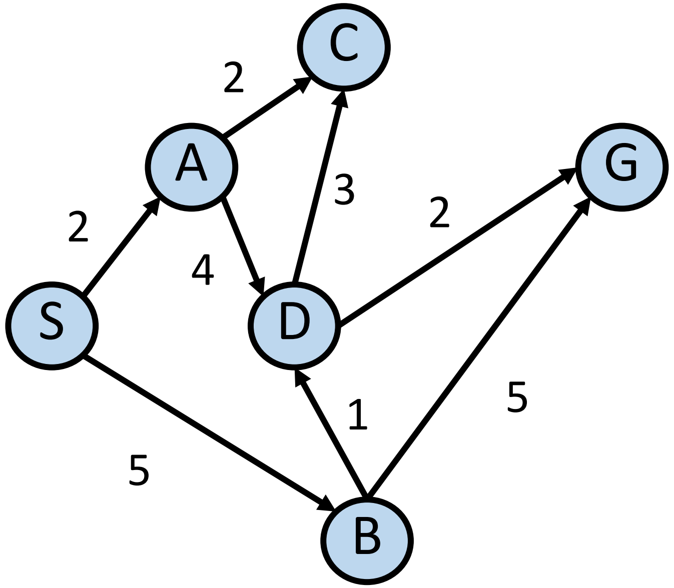
State	Cost
$\langle D, C, A, S \rangle$	7
$\langle G, D, A, S \rangle$	8
$\langle G, B, S \rangle$	10



Example of Uniform-Cost Search

Queue

State	Cost
$\langle G, D, A, S \rangle$	8
$\langle G, D, C, A, S \rangle$	9
$\langle G, B, S \rangle$	10



Remarks on Uniform Cost Search (UCS)

- UCS is an extension of BFS to the weighted-graph case (UCS = BFS if all edges have the same cost).

- UCS is complete and optimal (assuming costs bounded away from zero).

- UCS is guided by path cost rather than path depth, so it may get in trouble if some edge costs are very small.

- Worst-case time and space complexity $O\left(b^{\frac{W^*}{\epsilon}}\right)$, where W^* is the optimal cost, and ϵ is such that all edge weights are no smaller than ϵ.

Greedy (Best-First) Search

- UCS explores paths in all directions, with no bias towards the goal state.
- What if we try to get "closer" to the goal?
- We need a measure of distance to the goal. It would be ideal to use the length of the shortest path... but this is what we are trying to compute!
- We can estimate the distance to the goal through a **"heuristic function,"** $h : V \rightarrow \mathbb{R}_{\geq 0}$. In motion planning, we can use, e.g., the Euclidean distance to the goal (as the crow flies).
- A reasonable strategy is to always try to move in such a way to minimize the estimated distance to the goal: this is the basic idea of the **greedy (best-first) search**.

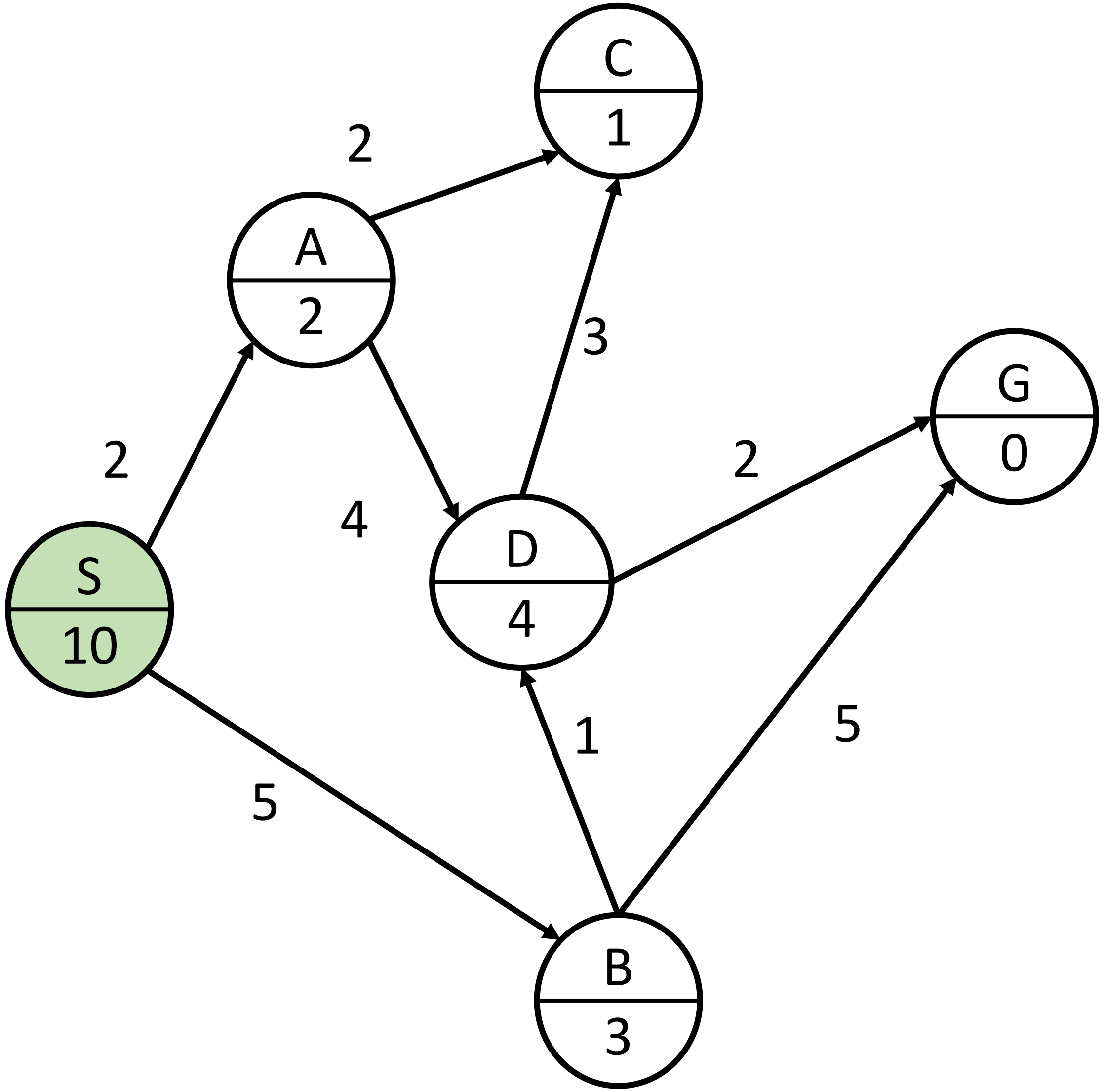
Greedy (Best-First) Search

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N   //Goal reach!
4. Else:
   a) Remove partial path N with the lowest cost $\boldsymbol{h(\mathbf{head}(N))}$ from the queue Q
   b) Find all children of head(N) and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) GOTO step 2

Example of Greedy (Best-First) Search

Queue

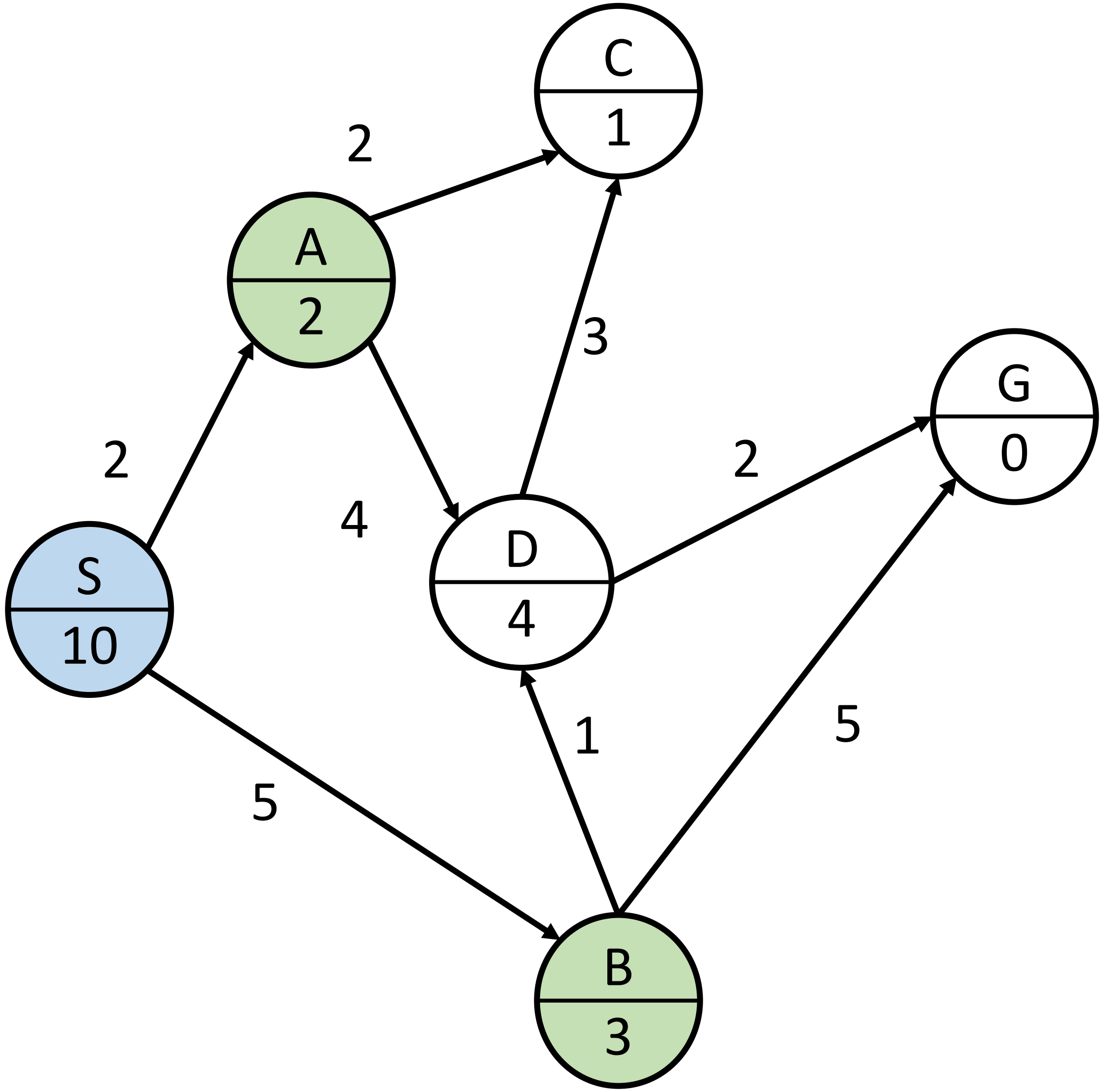
State	Cost	h
$\langle S \rangle$	0	10



Example of Greedy (Best-First) Search

Queue

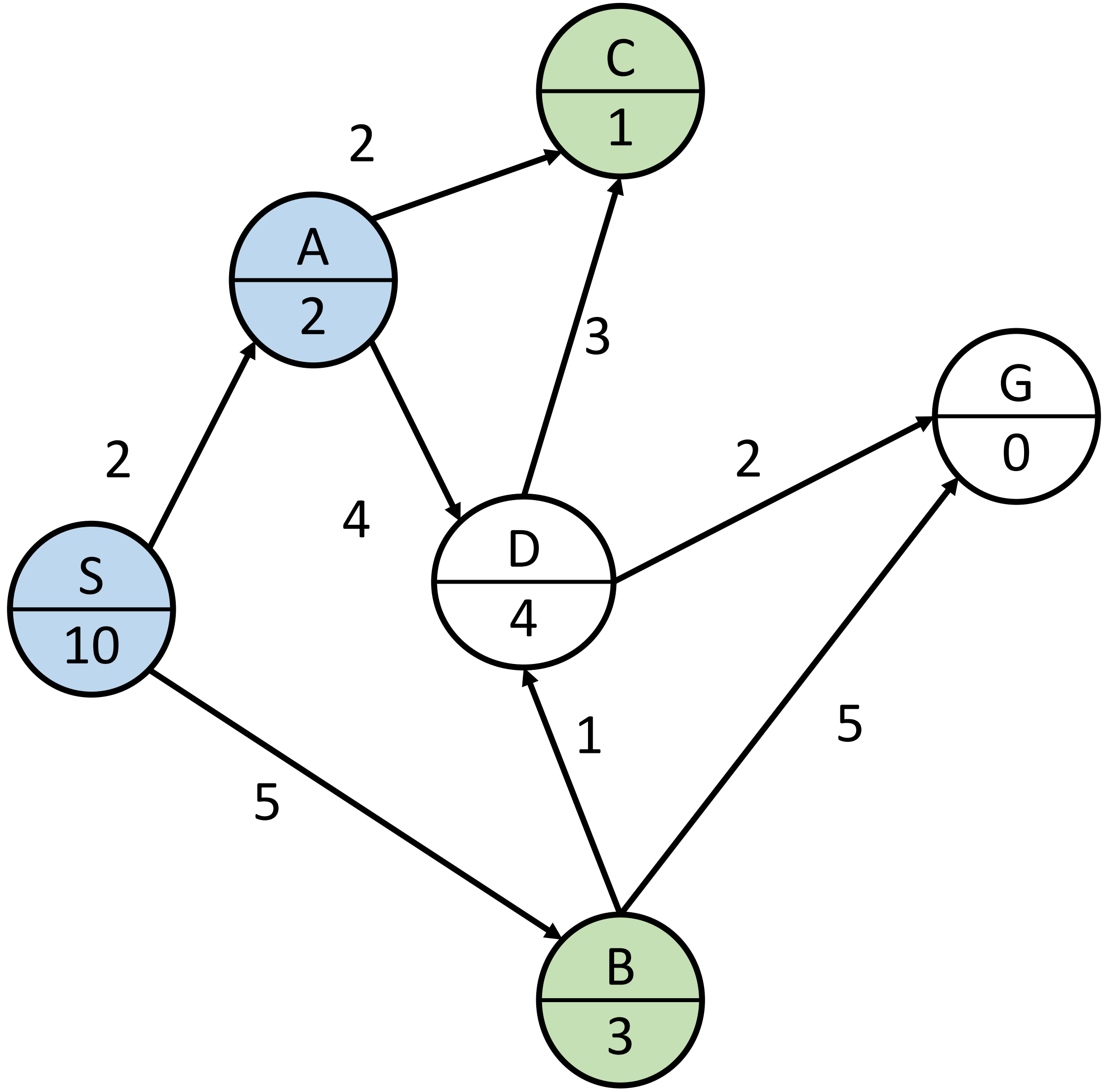
State	Cost	h
$\langle A, S \rangle$	2	2
$\langle B, S \rangle$	5	3



Example of Greedy (Best-First) Search

Queue

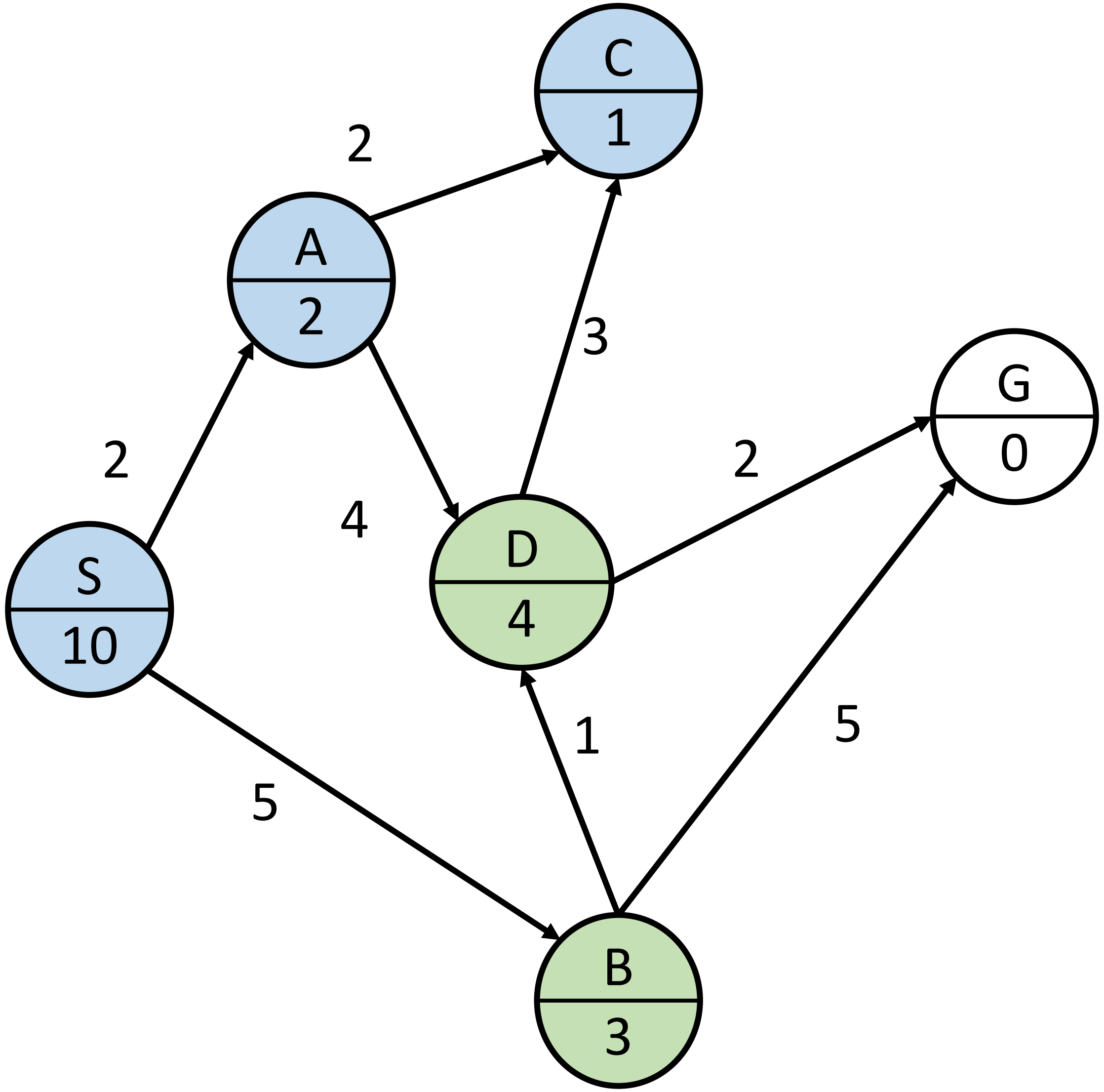
State	Cost	h
$\langle C, A, S \rangle$	4	1
$\langle B, S \rangle$	5	3
$\langle D, A, S \rangle$	6	4



Example of Greedy (Best-First) Search

Queue

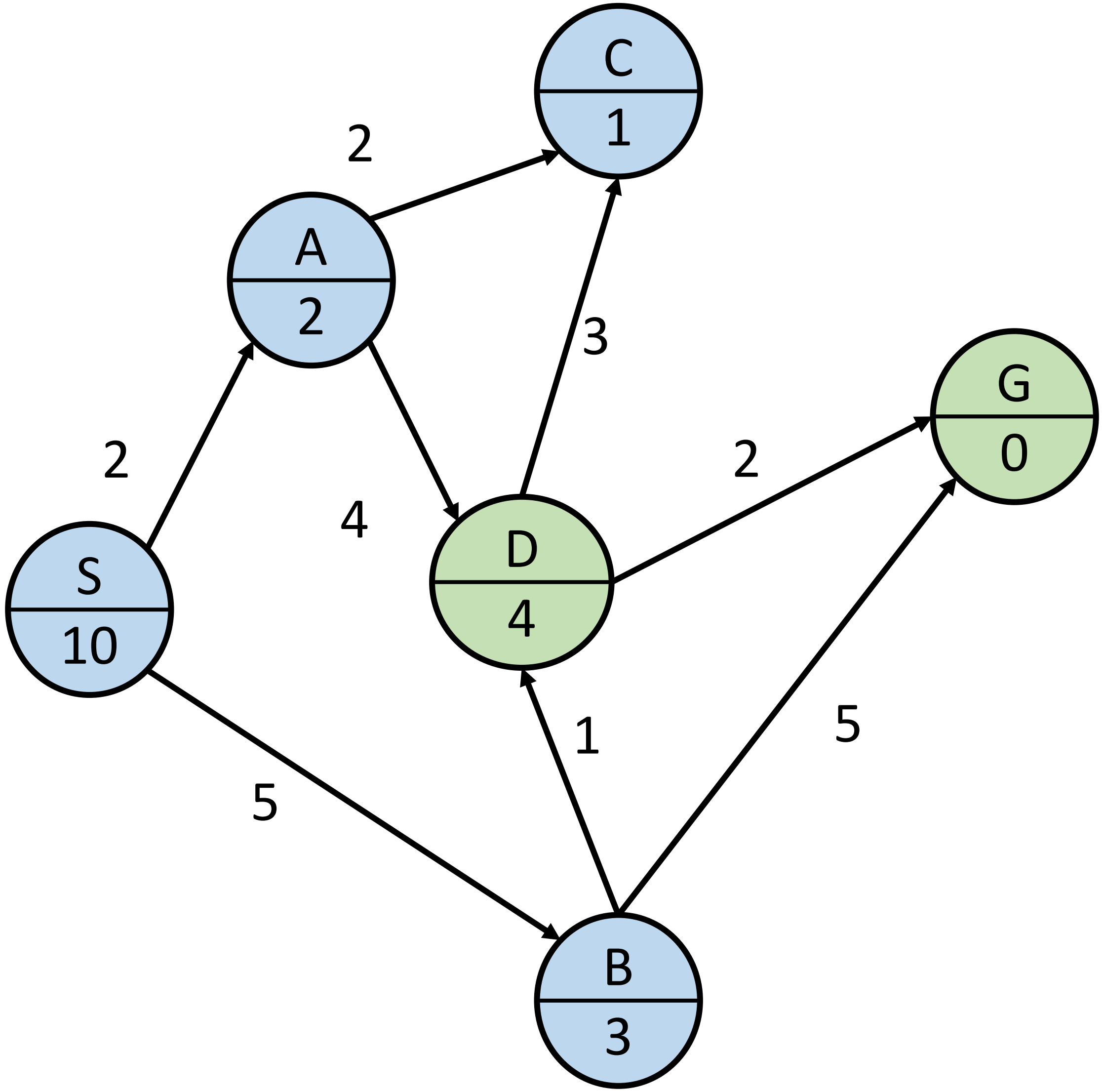
State	Cost	h
$\langle B, S \rangle$	5	3
$\langle D, A, S \rangle$	6	4
$\langle D, C, A, S \rangle$	7	4



Example of Greedy (Best-First) Search

Queue

State	Cost	h
$\langle G, B, S \rangle$	10	0
$\langle D, A, S \rangle$	6	4
$\langle D, C, A, S \rangle$	7	4



Remarks on Greedy (Best-First) Search

- Greedy (Best-First) search is similar in spirit to Depth-First Search: it keeps exploring until it has to back up due to a dead end.

- Greedy search is not complete and not optimal, but is often fast and efficient, depending on the heuristic function h.

- Worst-case time and space complexity $O(b^m)$.

Outline

- Informed search methods: Introduction
  - Shortest Path Problems on Graphs
  - Uniform-cost search
  - Greedy (Best-First) Search
- Optimal Search
- Monte-Carlo Tree Search

The A* Search Algorithm

The Problems:

- Uniform-Cost search is optimal, but may wander before finding the goal.
- Greedy search is not optimal, but in some cases it is efficient, as it is heavily biased towards moving towards the goal. The non-optimality comes from neglecting “the past.”

The A* Search Algorithm

The Problems:

- Uniform-Cost search is optimal, but may wander before finding the goal.
- Greedy search is not optimal, but in some cases it is efficient, as it is heavily biased towards moving towards the goal. The non-optimality comes from neglecting “the past.”

The Idea:

- Keep track of cost of partial path to reach vertex, g(v), and the heuristic function estimating cost to reach goal from vertex, h(v).
- In other words, choose as a “ranking” function the sum of the two costs:

$$f(v) = g(v) + h(v)$$

- g (v): cost-to-come (from the start to v).
- h(v): cost-to-go estimate (from v to the goal).
- f (v): estimated cost of the path (from the start to v and then to the goal).

Greedy (Best-First) Search

1. Initialize Q with a partial path <S>; set Visited = {}
2. If Q is empty, fail. Else, pick a partial path N from Q
3. If head(N) = G, return N //Goal reach!
4. Else:
   a) Remove partial path N with the lowest **estimated** $\mathbf{f(N) = g(N) + h(head(}\boldsymbol{N}\mathbf{))}$ from the queue Q
   b) Find all children of head(N) and create a on-step extension of N to each child
   c) Add all extended paths to Q
   d) Add Children of head(N) to Visited
   e) GOTO step 2

$$f(N) = g(N) + h(head(N))$$

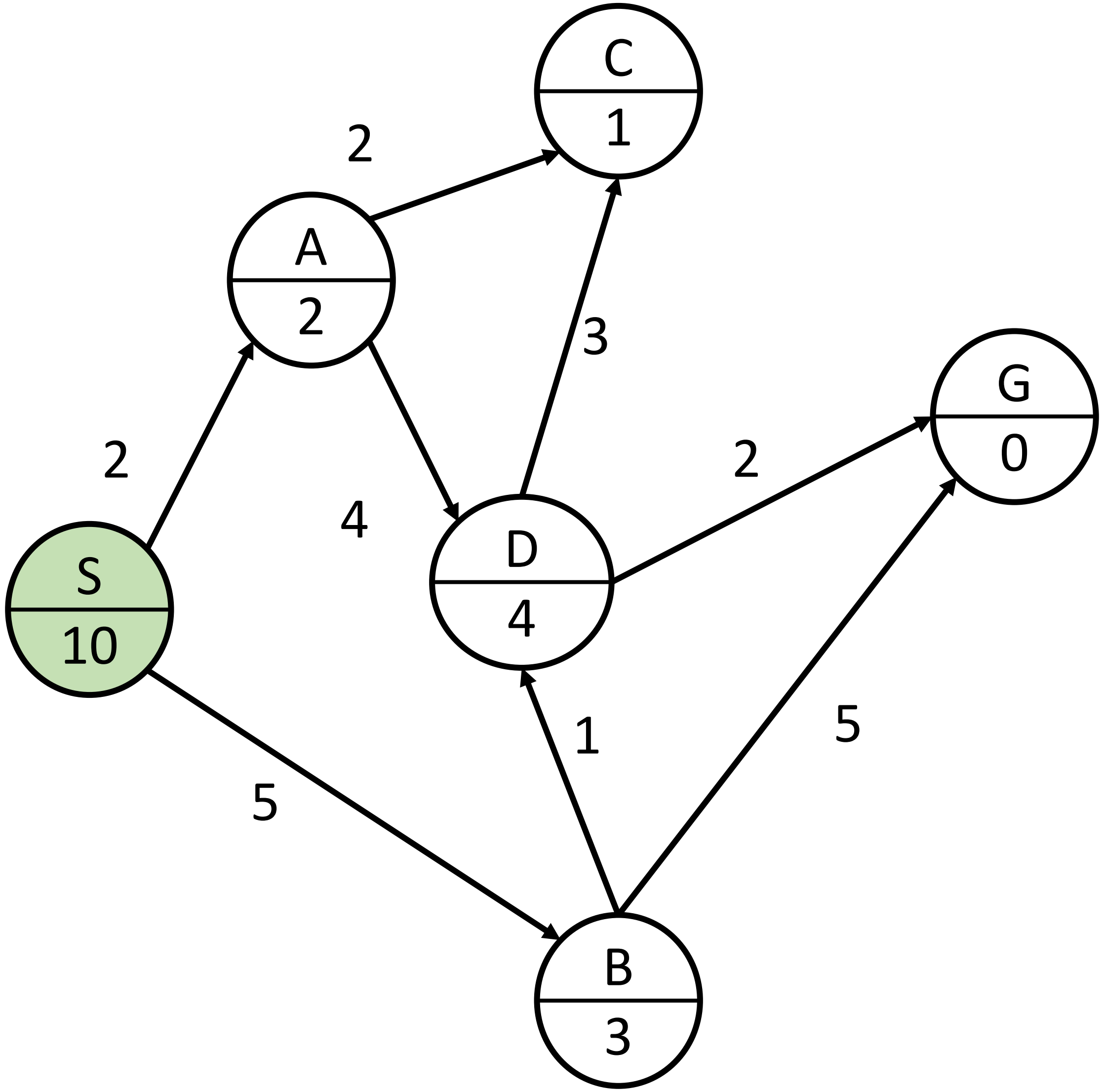
$g(v)$: cost thus far

$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle S \rangle$	0	10	10

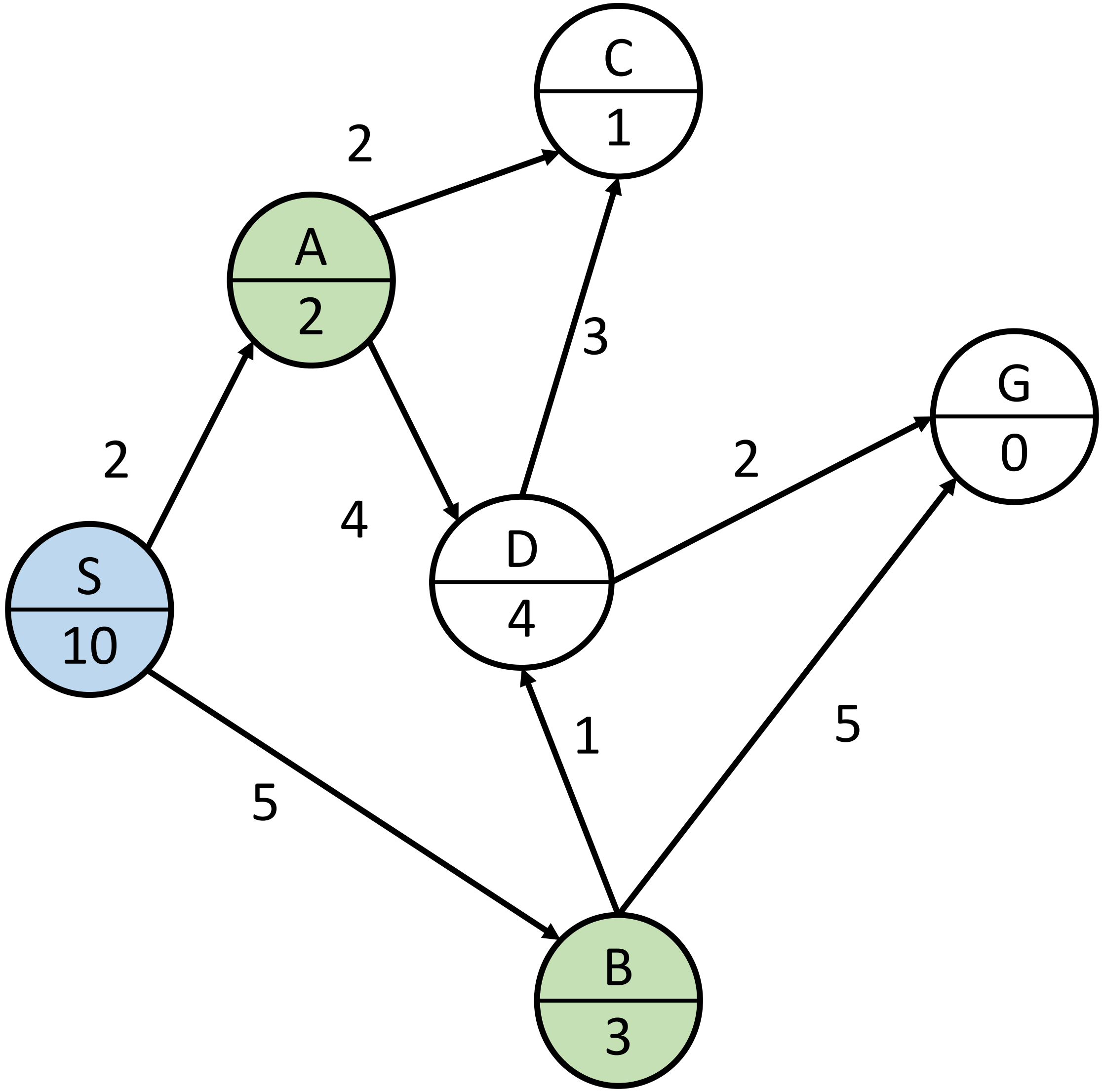


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle A, S\rangle$	2	2	4
$\langle B, S\rangle$	5	3	8

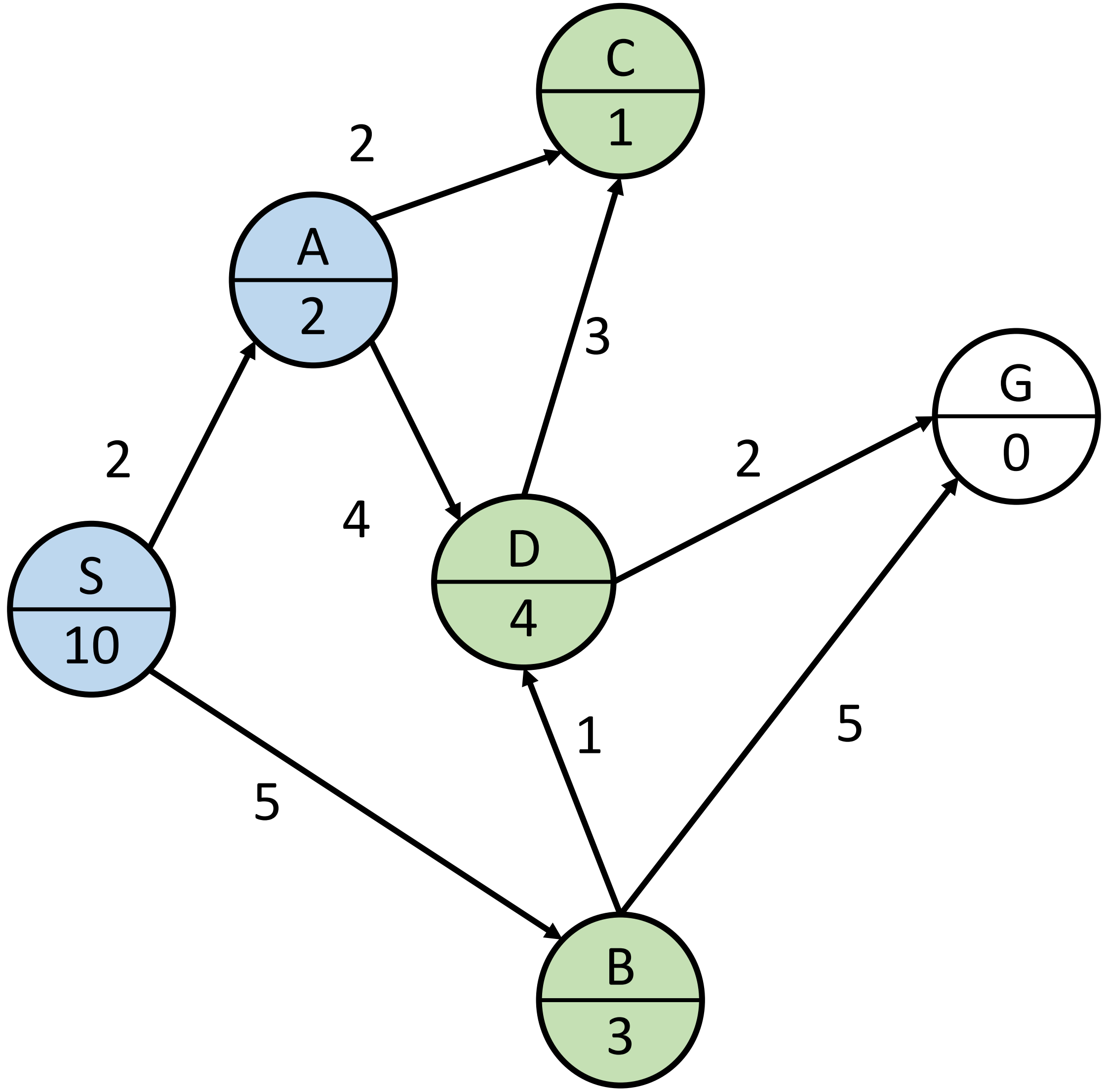


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle C, A, S \rangle$	4	1	5
$\langle B, S \rangle$	5	3	8
$\langle D, A, S \rangle$	6	5	11

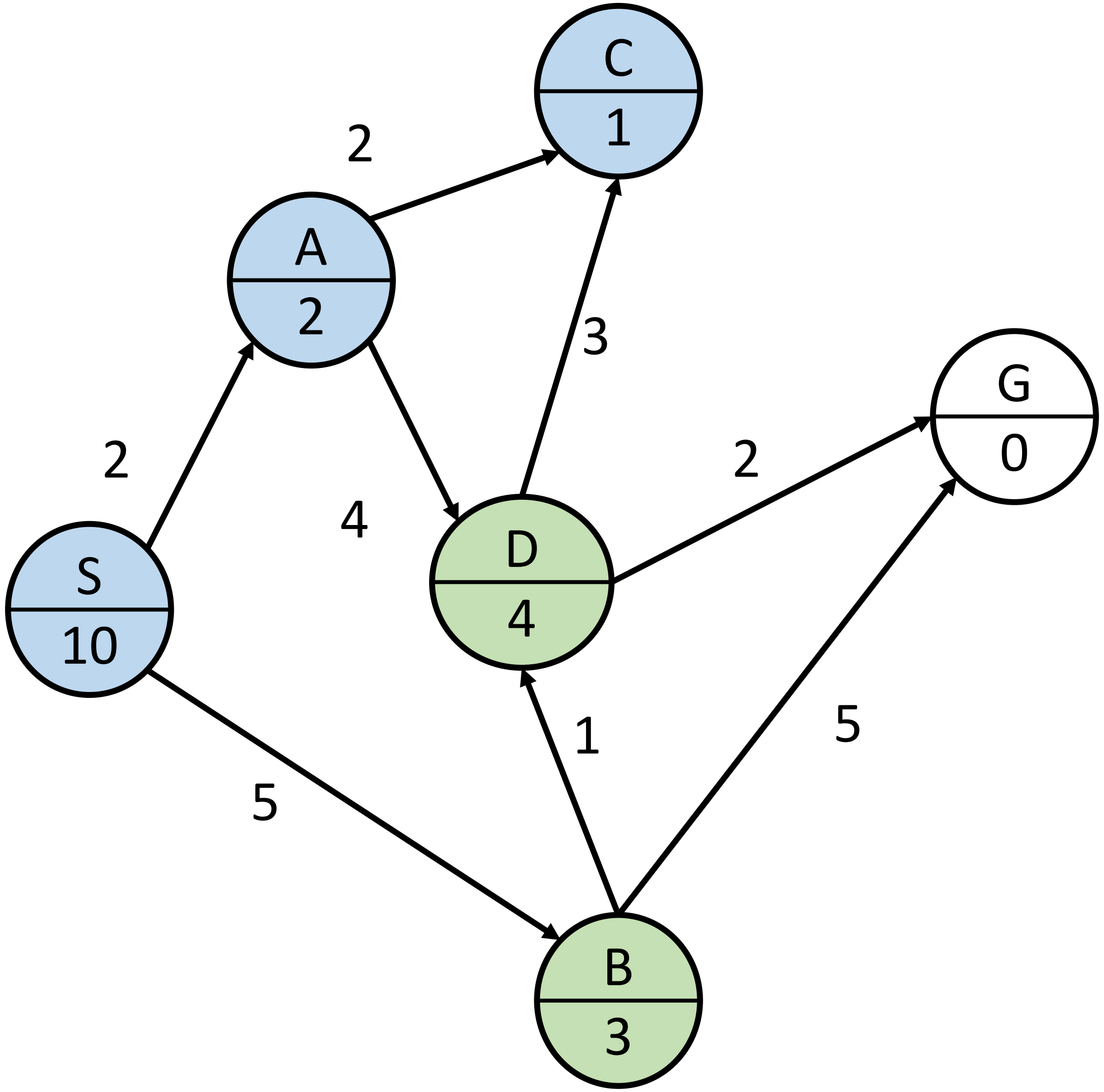


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle B, S \rangle$	5	3	8
$\langle D, A, S \rangle$	6	5	11
$\langle D, C, A, S \rangle$	7	5	12

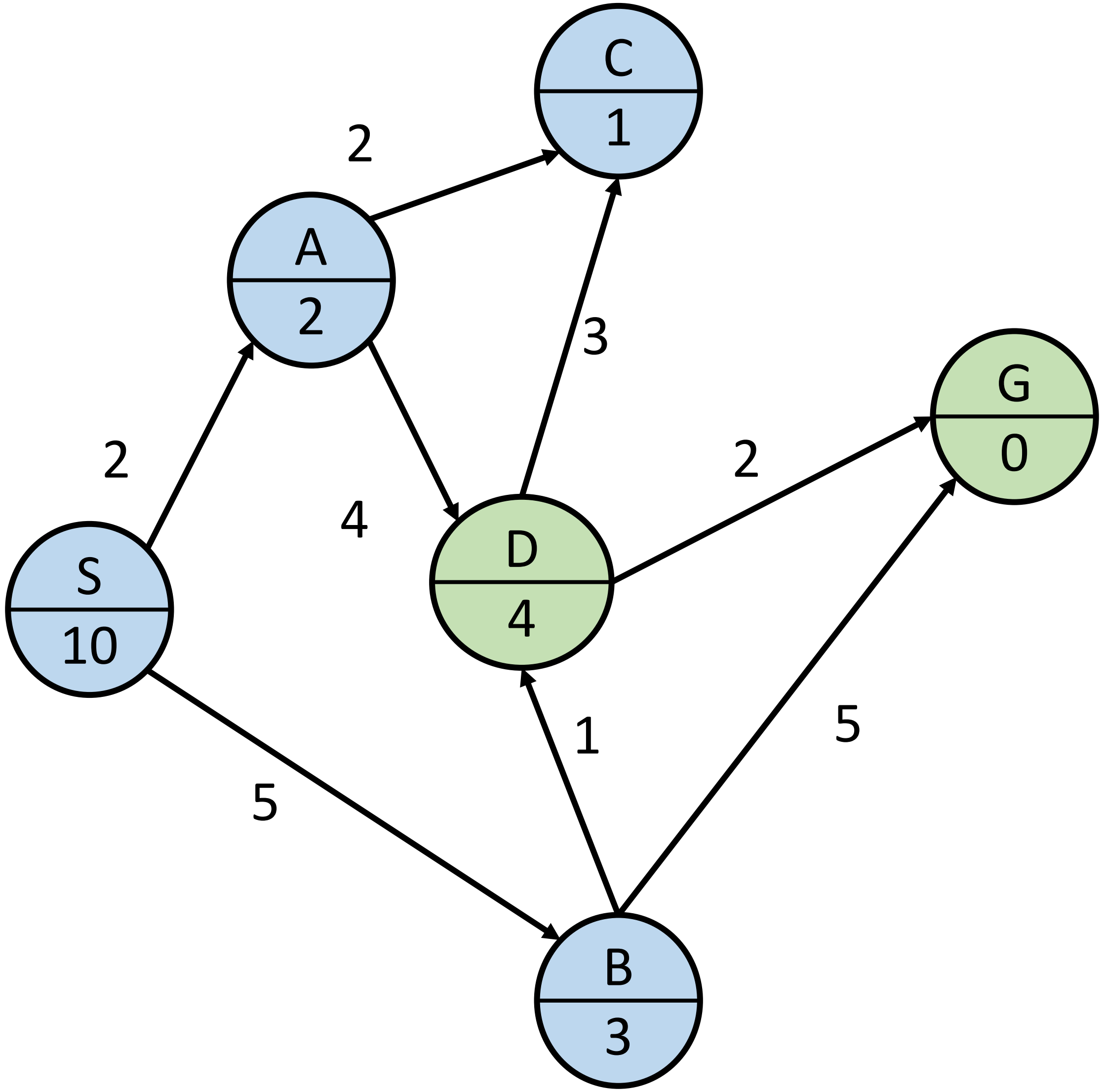


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle G, B, S \rangle$	10	0	10
$\langle D, A, S \rangle$	6	5	11
$\langle D, C, A, S \rangle$	7	5	12



Remarks on the A* Search algorithm

- A* Search is similar to UCS, with a bias induced by the heuristic h. If h = 0, A = UCS.
- The A* Search is complete, but is not optimal. What is wrong? (Recall that if h = 0, then A = UCS, and hence optimal...)

A* Search

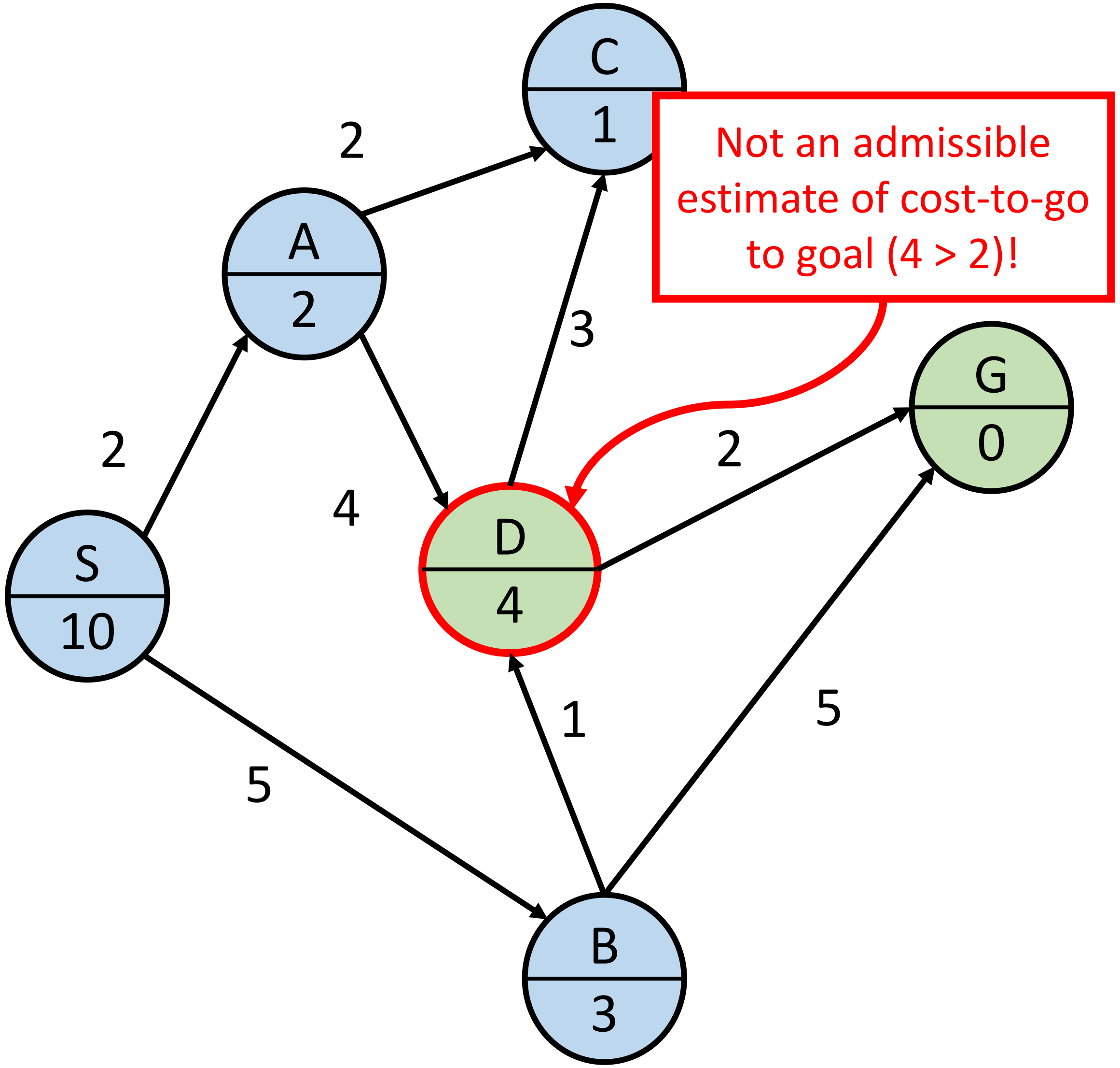
- Choose an admissible heuristic, i.e., such that $h(v) \leq h^*(v)$. (The star means "optimal.")
- The *A* Search* with an admissible heuristic is called A^*, which is guaranteed to be optimal.

$$f(N) = g(N) + h(head(N))$$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle G, B, S \rangle$	10	0	10
$\langle D, A, S \rangle$	6	5	11
$\langle D, C, A, S \rangle$	7	5	12

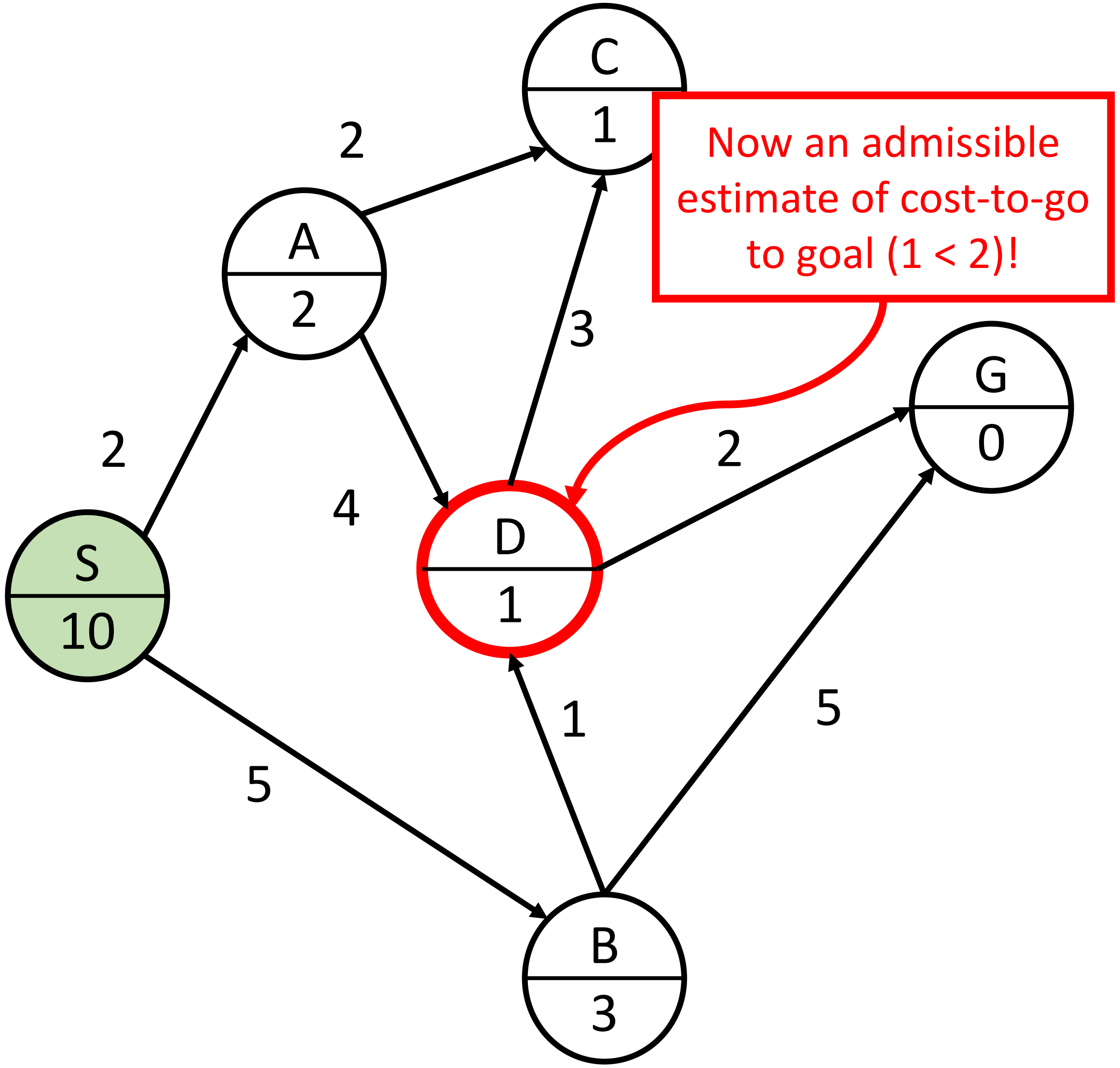


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle S \rangle$	0	10	10

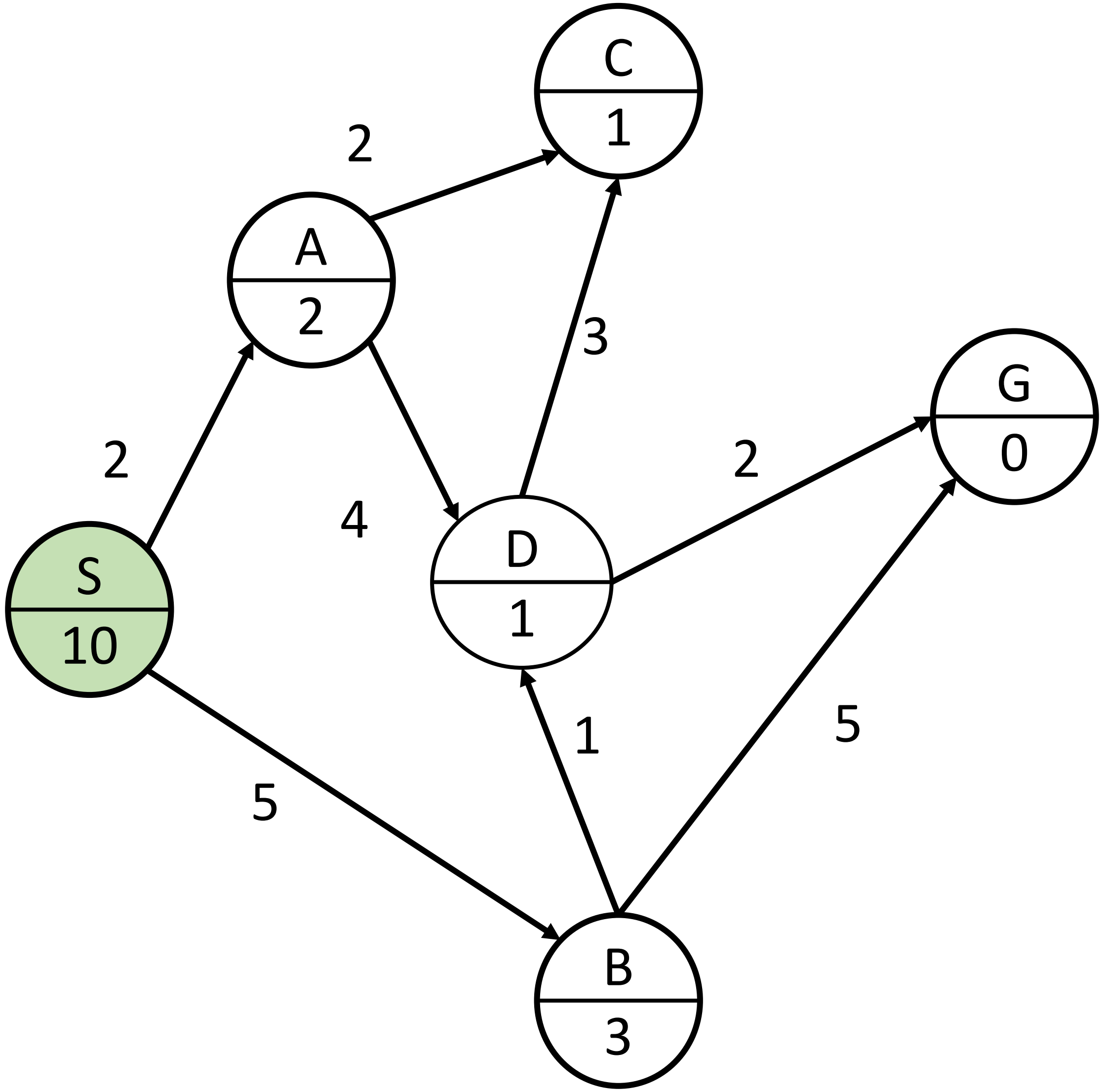


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle S \rangle$	0	10	10

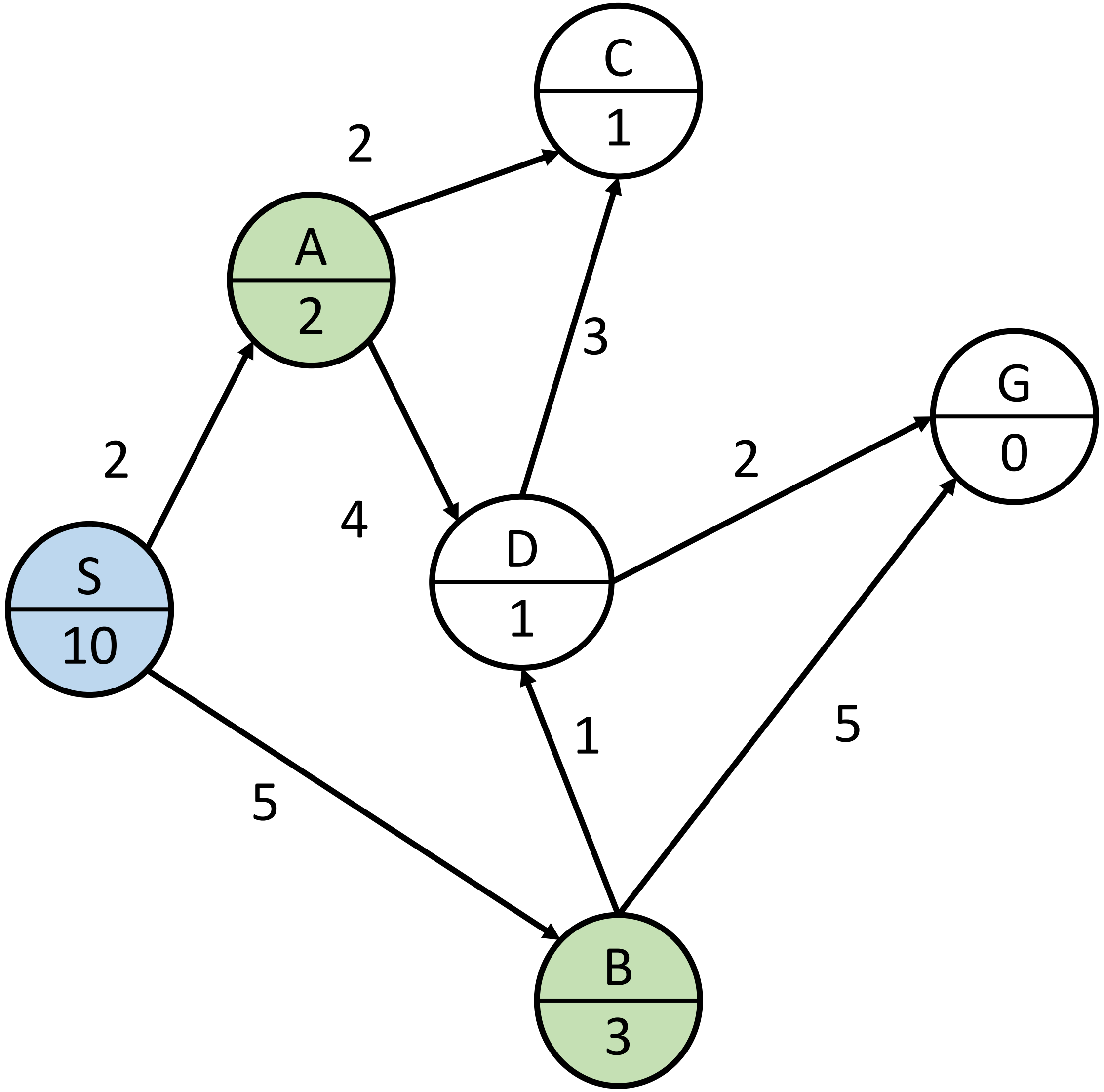


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle A, S \rangle$	2	2	4
$\langle B, S \rangle$	5	3	8

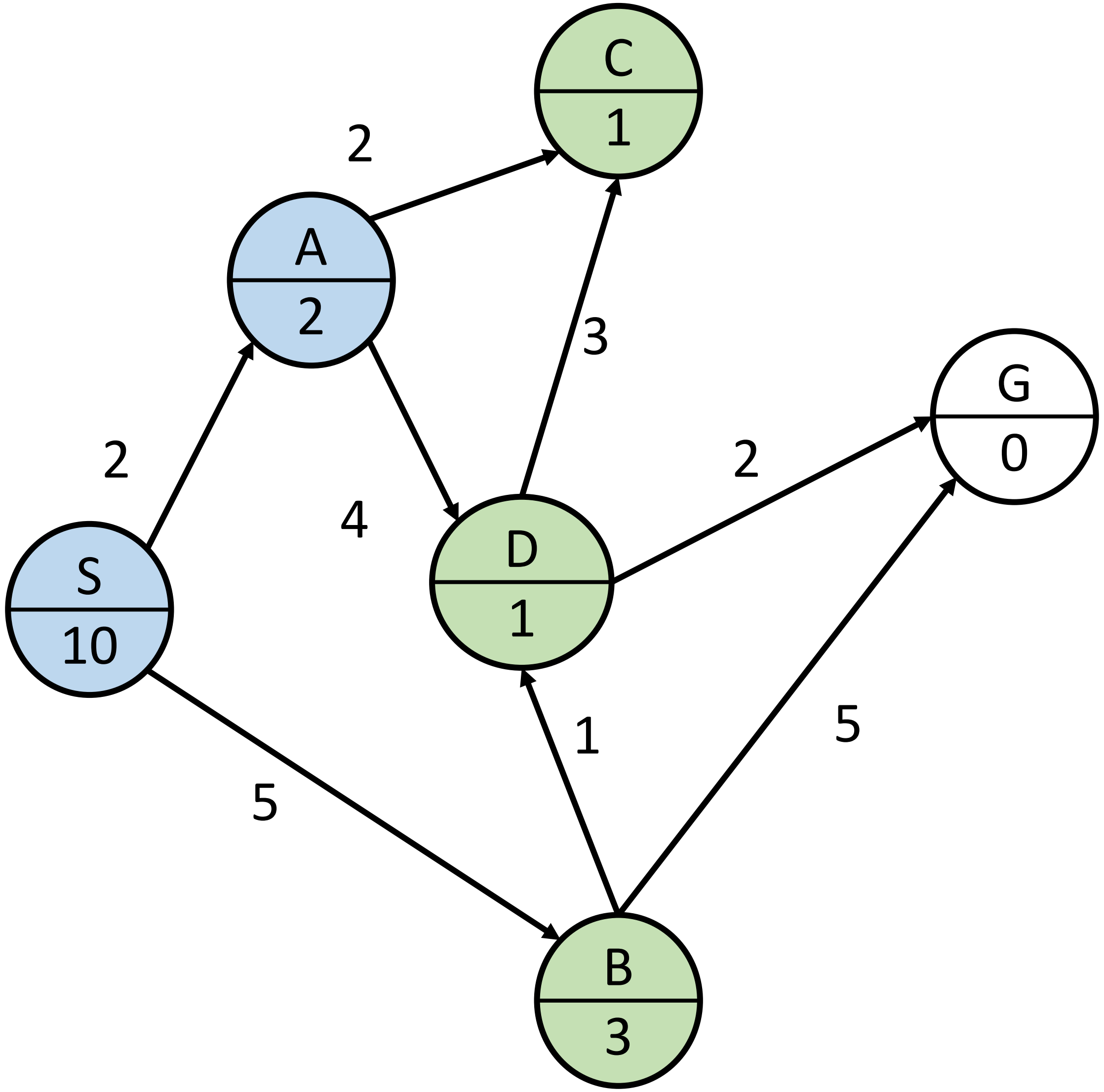


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle C, A, S \rangle$	4	1	5
$\langle D, A, S \rangle$	6	1	7
$\langle B, S \rangle$	5	3	8

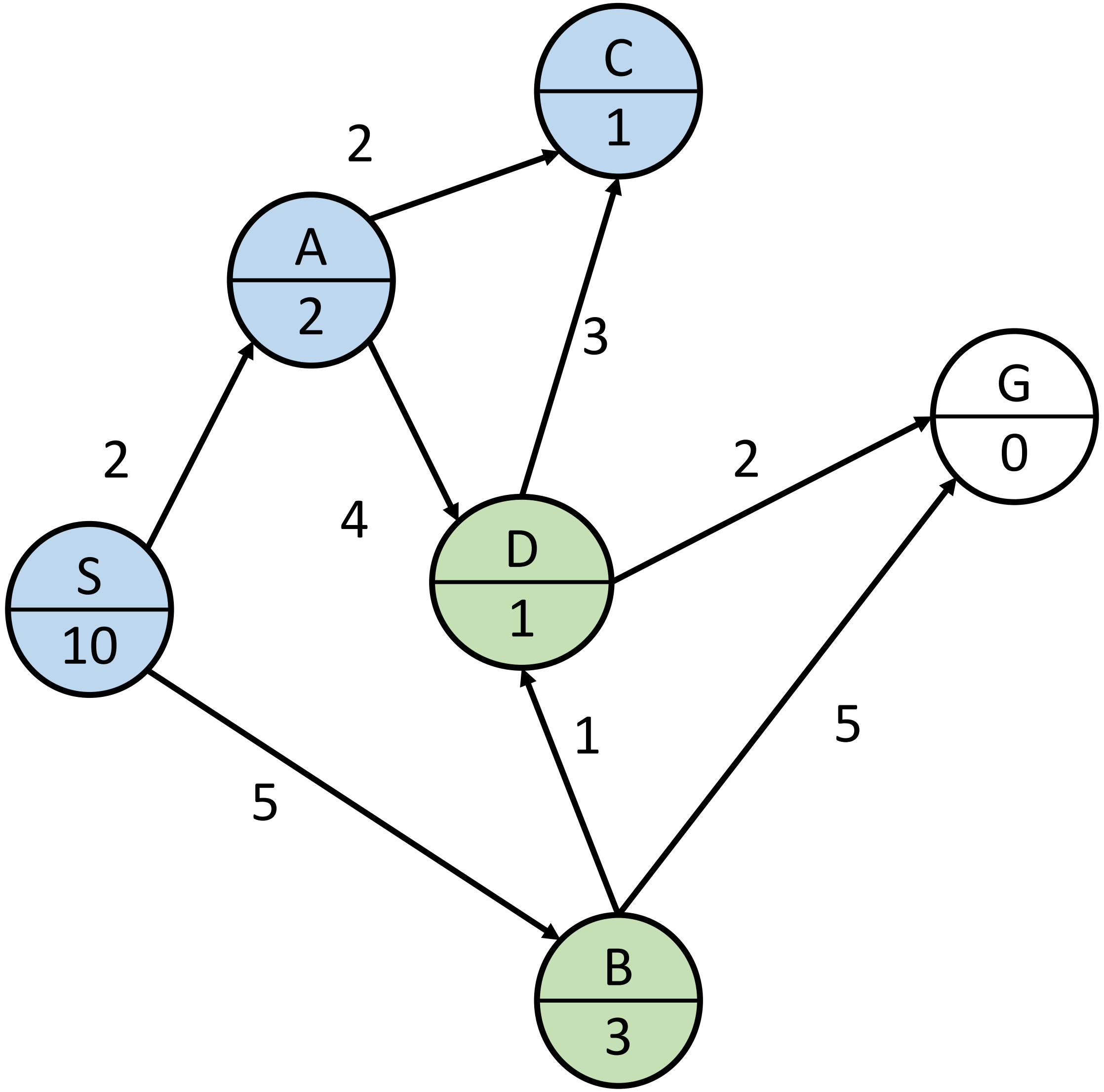


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle D, A, S \rangle$	6	1	7
$\langle B, S \rangle$	5	3	8
$\langle D, C, A, S \rangle$	7	1	8

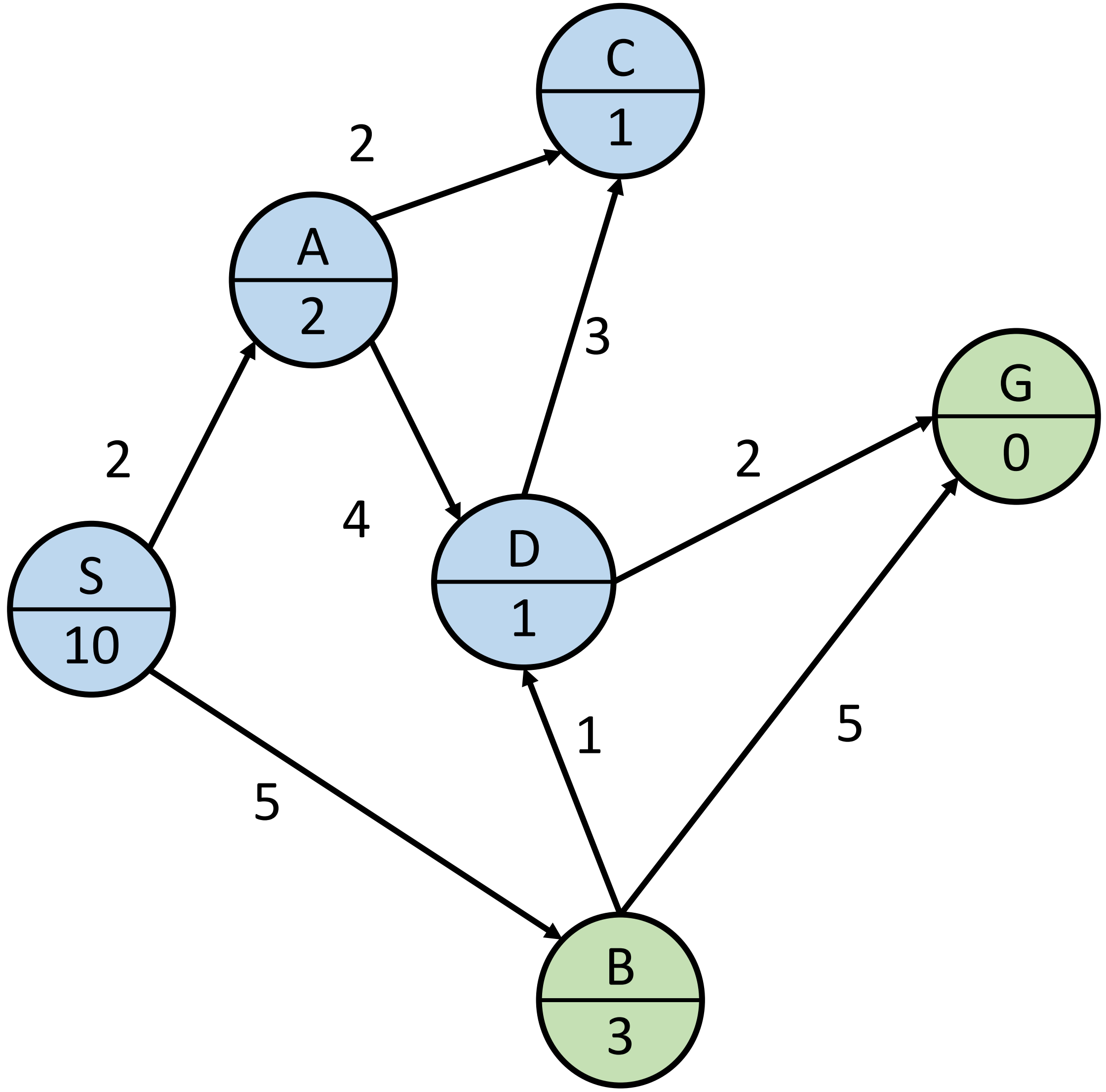


$f(N) = g(N) + h(head(N))$
$g(v)$: cost thus far
$h(v)$: cost-to-go

Example of Greedy (Best-First) Search

Queue

State	g	h	f
$\langle G, D, A, S \rangle$	8	0	8
$\langle B, S \rangle$	5	3	8
$\langle D, C, A, S \rangle$	7	1	8



Proof (sketch) of A* optimality

By Contradiction:

- Assume that A* returns P, but $w(P) > w^*$ (w^* is the optimal path weight/cost)
- Find the first unexpanded node on the optimal path P*, call it n
- $f(n) > w(P)$, otherwise, we would have expanded n
- $f(n) = g(n) + h(n)$ by definition
- $= g^*(n) + h(n)$ because n is on the optimal path
- $\leq g^*(n) + h^*(n)$ because h is admissible
- $= f^*(n) = W^*$ because h is admissible
- Hence, $W^* \geq f(n) > W$, which is a contradiction

Admissible heuristics

- How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go

Admissible heuristics

- How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go

Examples:

- h(v) = 0: This always works; however, not useful as it is just A* = UCS
- h(v) = distance(v,g) when vertices of graph are physical locations
- h(v) = $|\mathrm{v}-\mathrm{g}|_p$, when the vertices of the graphs are points in a p-normed vector space

Admissible heuristics

- How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go

Examples:

- h(v) = 0: This always works; however, not useful as it is just A* = UCS
- h(v) = distance(v,g) when vertices of graph are physical locations
- h(v) = $|\mathrm{v} - \mathrm{g}|_p$, when the vertices of the graphs are points in a p-normed vector space

General method:

- Choose h as the optimal cost-to-go function for a relaxed problem, that is easy to compute

(Relaxed problem: ignore some of the constraints in the original problem)

Admissible heuristics for the 8-puzzle

5	4	
6	1	8
7	3	2

Start

1	2	3
8		4
7	6	5

Goal

Which of the following are admissible heuristics?

- h = 0
- h = 1
- h = # tiles in wrong position
- h = sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle

5	4	
6	1	8
7	3	2

Start

1	2	3
8		4
7	6	5

Goal

Which of the following are admissible heuristics?

- h = 0 **YES**, always good
- h = 1
- h = # tiles in wrong position
- h = sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle

5	4	
6	1	8
7	3	2

Start

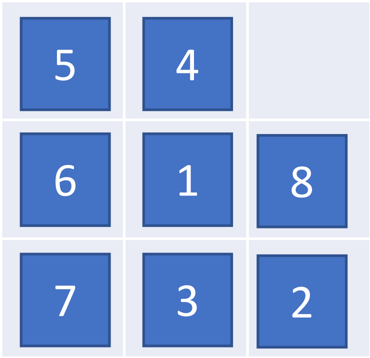
1	2	3
8		4
7	6	5

Goal

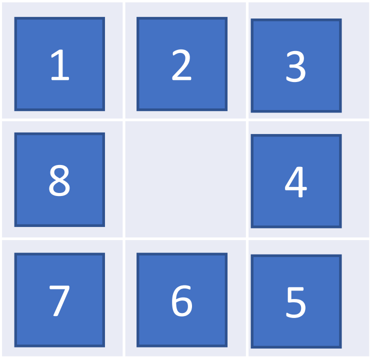
Which of the following are admissible heuristics?

- h = 0 **YES**, always good
- h = 1 **NO**, not valid in goal state
- h = # tiles in wrong position
- h = sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle



Start



Goal

Which of the following are admissible heuristics?

- h = 0 **YES**, always good
- h = 1 **NO**, not valid in goal state
- h = # tiles in wrong position **YES**, “teleport” each tile to goal in 1 move
- h = sum of (Manhattan) distance between tiles and their goal position.

Admissible heuristics for the 8-puzzle

5	4	
6	1	8
7	3	2

Start

1	2	3
8		4
7	6	5

Goal

Which of the following are admissible heuristics?

- h = 0 **YES**, always good
- h = 1 **NO**, not valid in goal state
- h = # tiles in wrong position **YES**, "teleport" each tile to goal in 1 move
- h = sum of (Manhattan) distance between tiles and their goal position.
  YES, move each tile to the goal ignoring other tiles

A partial order of heuristic functions

Some heuristics are better than others

- h = 0 is an admissible heuristic, but not very useful
- h = h* is also an admissible heuristic, and it is the “best” possible one as it gives us the optimal path directly with no search/backtracking

Partial order

- We can say that h_1 dominates h_2 if $\mathrm{h}_1(v) \geq \mathrm{h}_2(v)$ for all vertices v.
- h* dominates all admissible heuristics, and 0 is dominated by all admissible heuristics

Choosing the right heuristic

- In general, we want a heuristic that is as close to h* as possible. However, such a heuristic may be too complicated to compute. There is a tradeoff between complexity of computing h and the complexity of search

Consistent heuristics

- An additional useful property for A* heuristics is called consistency
- A heuristic $h: X \rightarrow \mathbb{R}^+$ is said consistent if

$$\mathrm{h(u) \leq w\big(e = (u, v)\big) + h(v), \forall (u, v) \in E}$$

- In other words, a consistent heuristic satisfies the triangle inequality
- If h is consistent heuristic, then f = g + h is non-decreasing along paths

$$f(v) = g(v) + h(v) = g(v) + w(u, v) + h(v) \geq f(u)$$

- Hence, the values of f on sequence of nodes expanded by A* is non-decreasing: First path found to node is also optimal path → no need to compare costs!

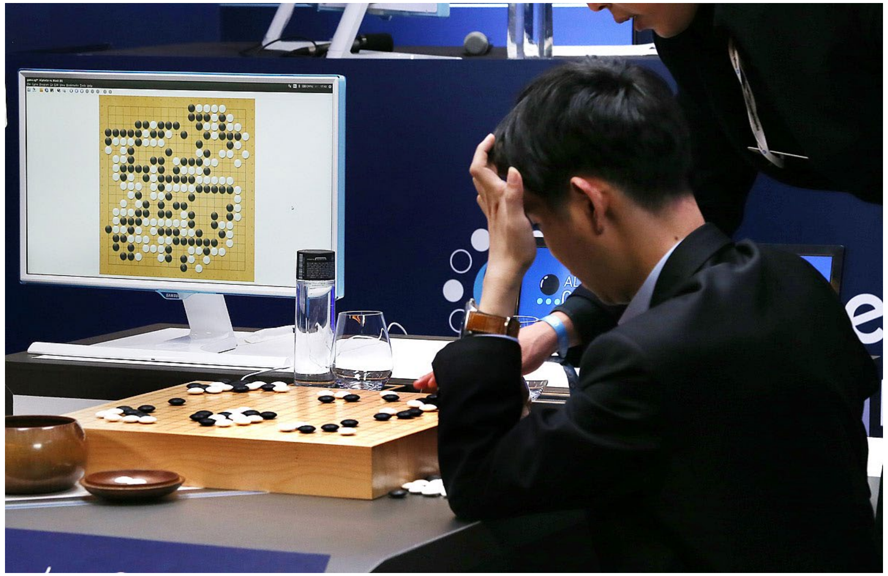
Mid-lecture break

Outline

- Informed search methods: Introduction
  - Shortest Path Problems on Graphs
  - Uniform-cost search
  - Greedy (Best-First) Search
- Optimal Search
- Monte-Carlo Tree Search

Why Monte Carlo Tree Search (MCTS)?

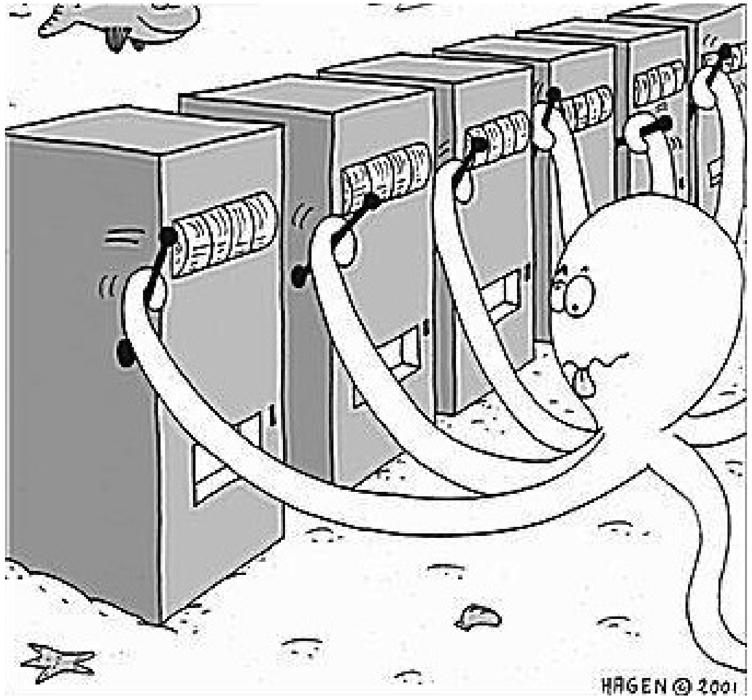
- Foundation of state-of-the-art gameplaying algorithms, e.g., AlphaGo
- Challenge:
  - Too complex to perform UCS
  - Too hard to "write down" an admissible heuristic
- Idea:
  - Before each decision, probabilistically generate a partial search tree
  - Cost-to-go based upon a DFS-like playing of game to completion

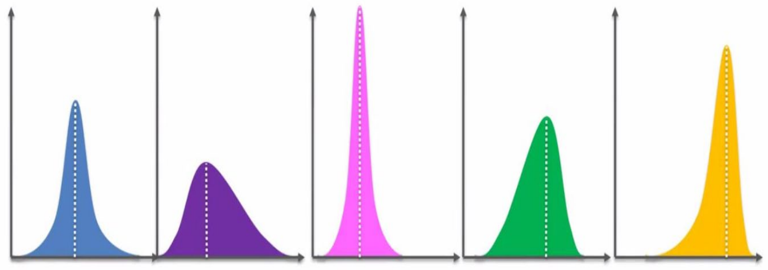


AlphaGo

Preliminaries: Multi-arm Bandit

- Given:
  - N “arms” that can be “pulled,” returning a reward drawn from an arm-specific random distribution
- Problem:
  - At each iteration, which arm to I pull to maximize my overall reward?
- Insight:
  - Do I keep pulling the current arm (exploitation) or try a different arm (exploration)?
- Metric:
  - Regret: the difference in reward you received versus what the optimal clairvoyant actor would have received from pulling the lever from the highest-pay-out arm





Preliminaries: UCB1

The algorithm UCB1 (Upper Confidence Bound 1), developed by Auer et al. (2002), is an algorithm for the multi-armed bandit that achieves regret that grows logarithmically with the number of actions taken

Concept:

- For each action j, record the average reward $\bar{x}_j$ and the number of times we have tried that action, n_j. n is the total number of actions $n = \sum_j n_j$.
- At each iteration, take action i that maximizes $\bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}$

MCTS Development

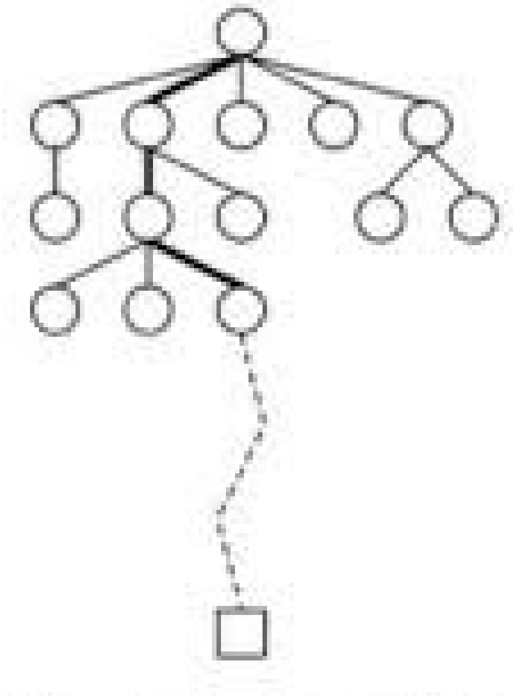
- Kocsis and Szepesvári, 2006
  - Describing bandit-based method
  - Simulate to approximate reward (searching full tree too hard)
  - Algorithm name: UCT (**UC**B applied to **T**rees)

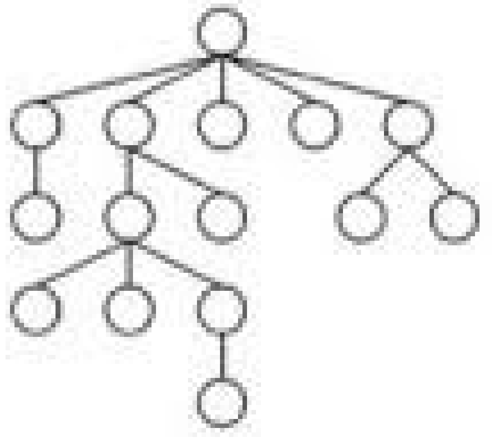
- UCT employs UCB1 algorithm on each explored node

- Proved MCTS converges to minimax solution

MCTS Overview

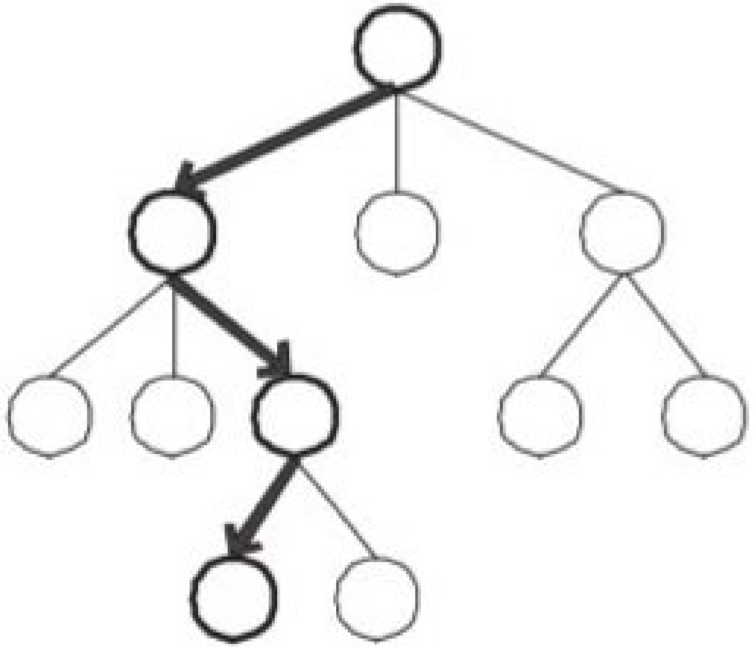
- Iteratively building partial search tree
- Iteration
  - Selection
    - Explore tree to a leaf
  - Expansion
    - Expand leaf and add child(ren)
  - Simulation
    - Evaluate “value” of child(ren)
  - Backpropagation
    - Move “value” backward to route





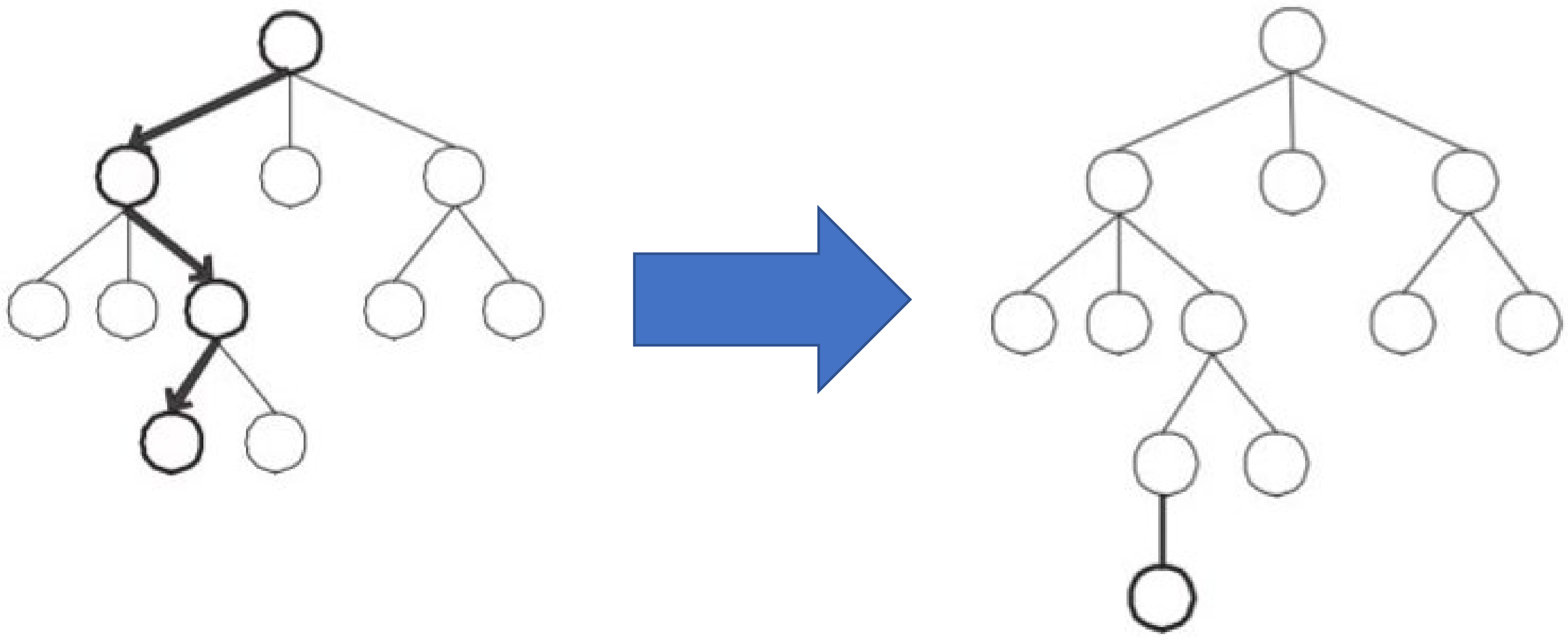
Step 1) Selection

- Start at root node (e.g., start node, current state)
- Based on <u>tree policy</u>, select child
- Apply recursively, descending through tree
  - Stop when expandable node is reached
  - Expandable: Node that is non-terminal (e.g., goal node) and has unexplored children



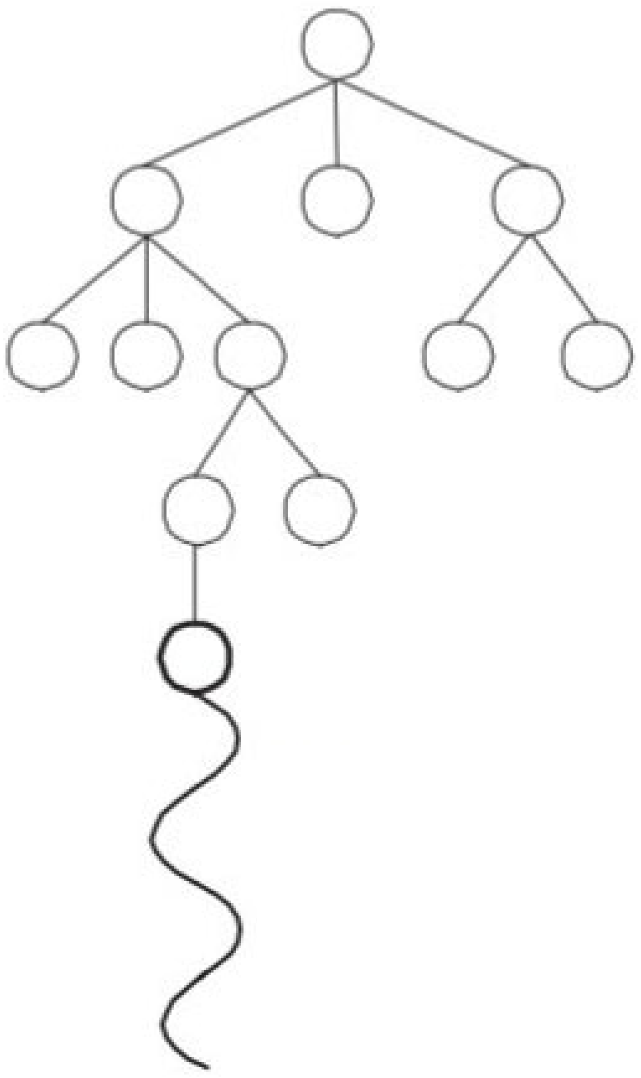
Step 2) Expansion

- Add one or more child nodes to tree
  - Depends on what actions are available for the current position
  - Method for choosing which child to add depends on <u>selection policy</u>
    - i.e., Tree policy chooses which child to add



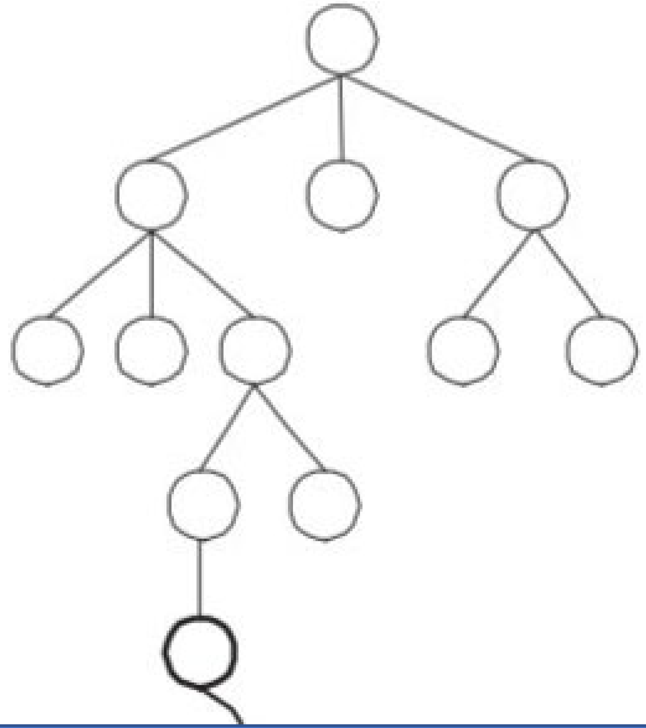
Step 3) Simulation

- Runs simulation of path that was selected
- Gets position at end of simulation
- Simulation policy determines how simulation is run
- Outcome determines value



Step 3) Simulation

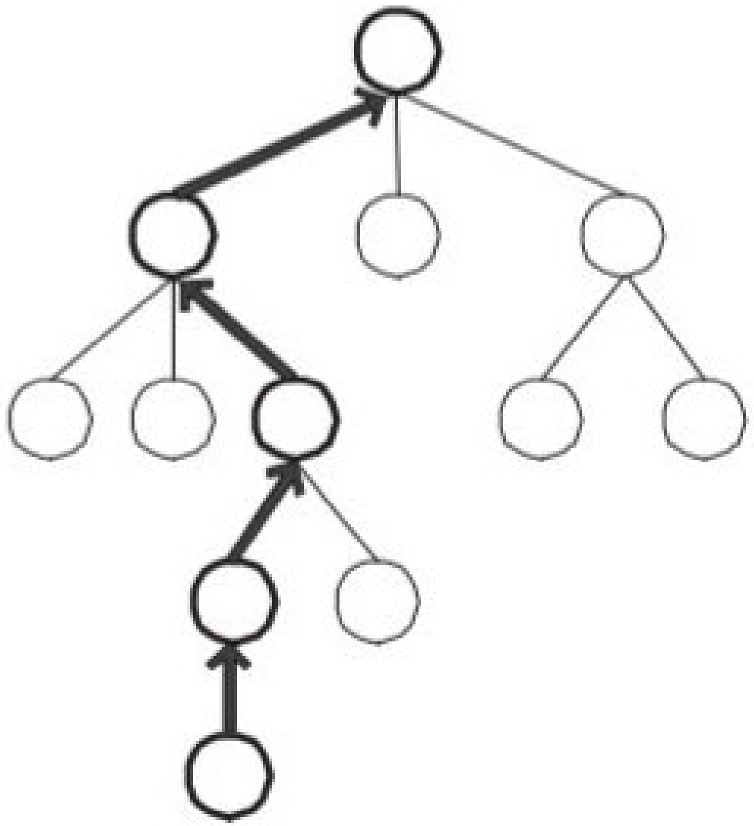
- Runs simulation of path that was selected
- Gets position at end of simulation
- Simulation policy determines how simulation is run
- Outcome determines value



If too costly to run to completion, could run partially to completion and use a heuristic evaluation (e.g., how many pieces are left to be captured) to guestimate the "likelihood" of winning, but only as good as your heuristic…

Step 4) Backpropagation

- Moves backward through saved path
- Value of node
  - Representative of benefit of going down that path from parent
- Values are updated dependent on outcome
  - Based on how the simulated game ends, values are updated



Policies

- Tree policy
- Selection policy
- Simulation policy

Often the same, as we'll see for AlphaGo

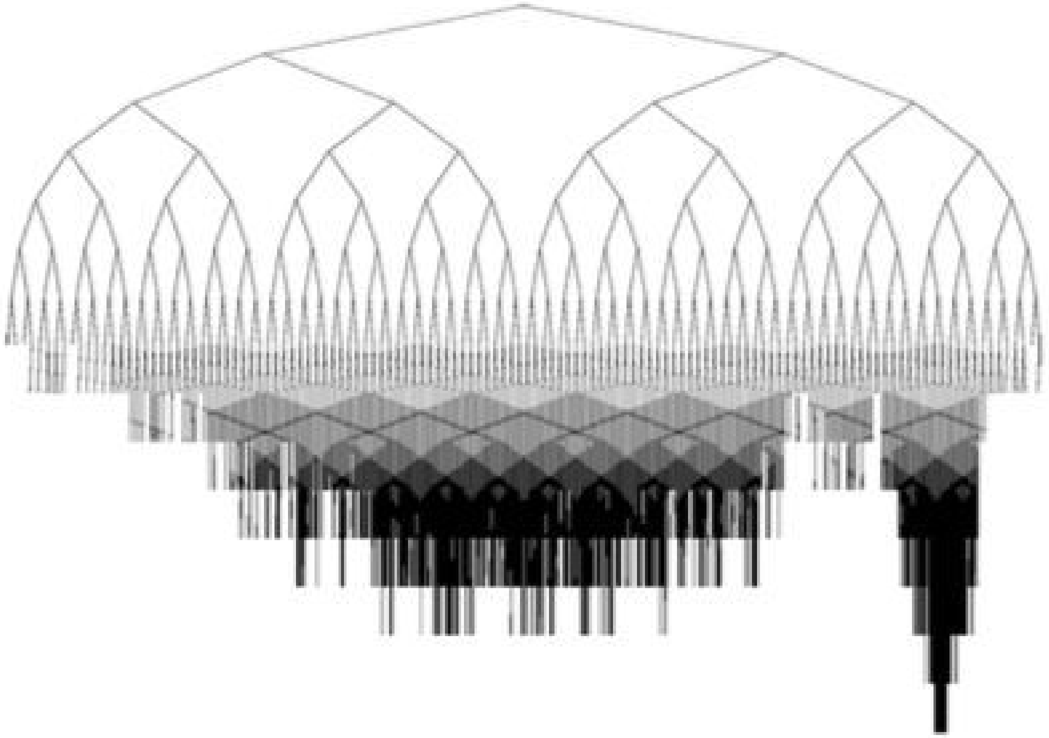
(revisit later)

Crux of UCT Algorithm

- Selecting child node is an instance of a multi-arm bandit problem
- Run UCB1 for each child selection (i.e., Tree Policy):
- $UCT = \bar{X}_j + 2C_p\sqrt{\frac{2 \ln n}{n_j}}$
  - n – number of times current (parent) node has been visited
  - n_j – number of times child j has been visited; $n_j = 0$ means infinite weight
  - C_p – some constant > 0, which is adjusted to control exploration vs. exploitation tradeoff
  - $\bar{X}_j$ – mean reward of selecting this position

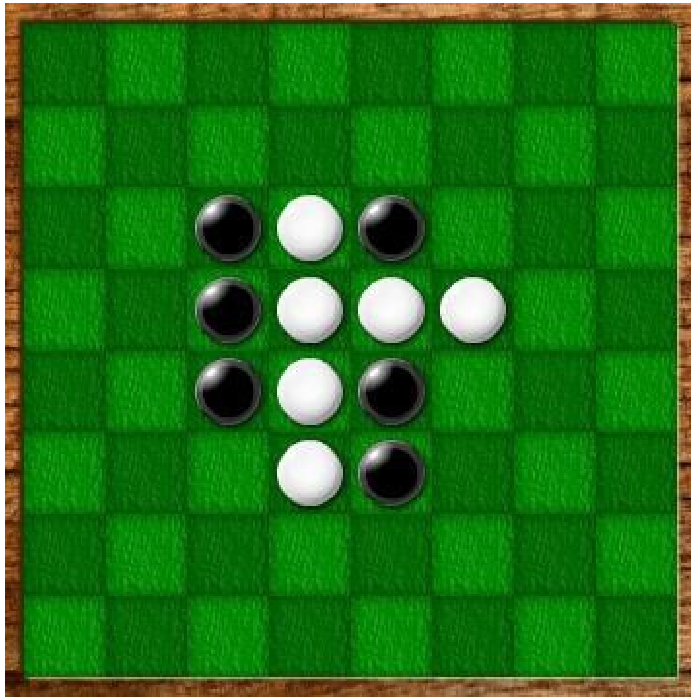
Pros/Cons of MCTS

- “Aheuristic” (i.e. ‘a’ prefix -> opposite)
  - No need for domain-specific knowledge
  - Other algorithms may work better if good heuristics exists
- Anytime
  - Can stop running MCTS at any time
  - Return best action
- Asymmetric
  - Favor more promising nodes
- Ramanujan et al.
  - Trap states = UCT performs worse
  - Can’t model sacrifices well (Queen Sacrifice in Chess)"



Example: Othello

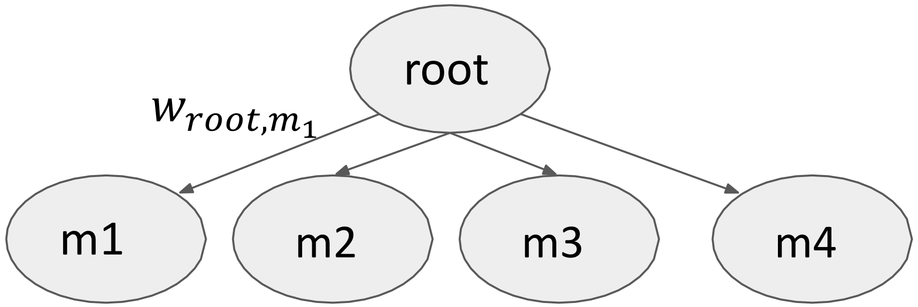
- Alternating turns
- You can only make a move that sandwiches a continuous line of your opponent's pieces between yours
- Color of sandwiched pieces switches to your color
- Ends when board is full
- Winner is whoever has more pieces

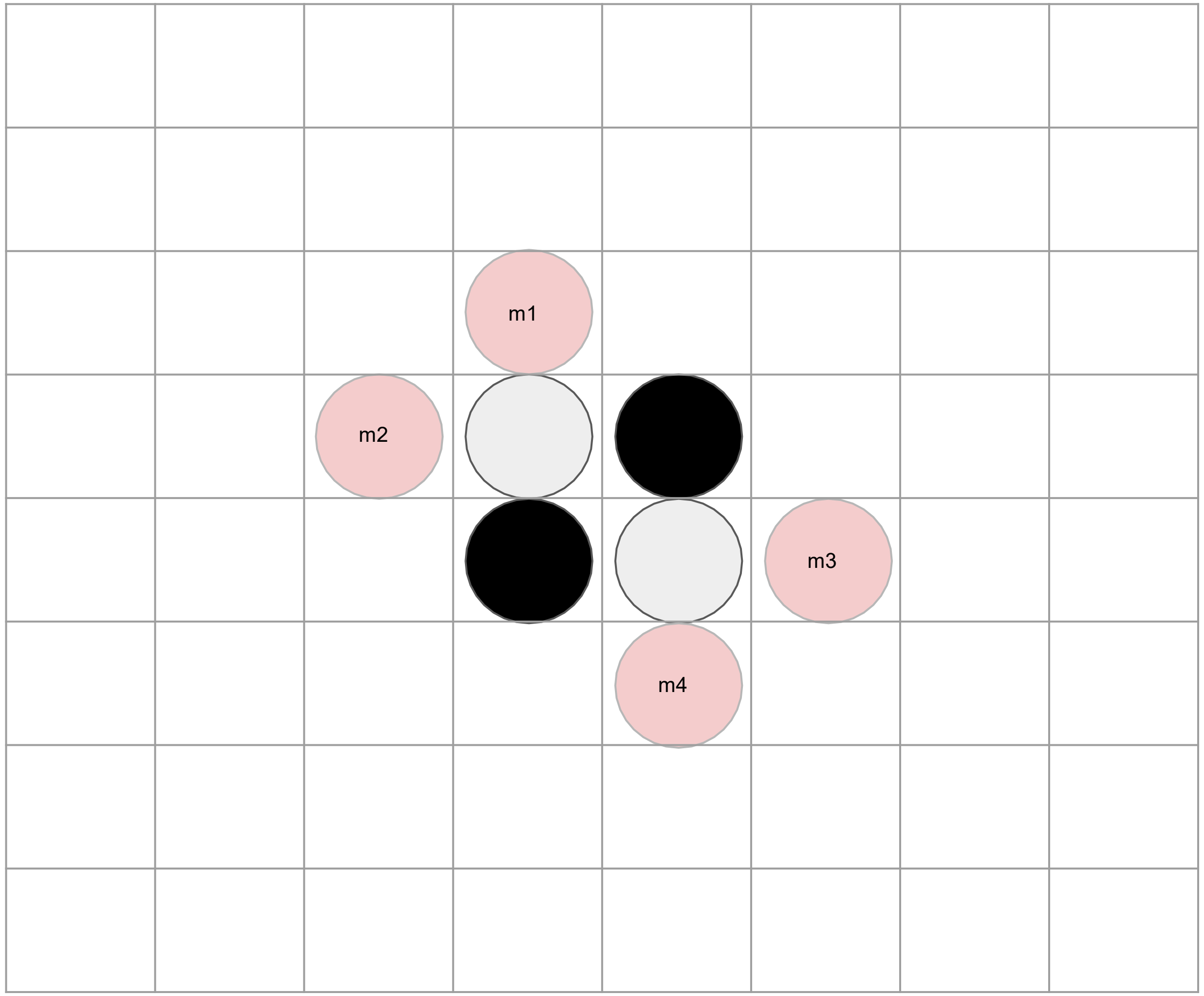


$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello





Initially:

- $n_j \leftarrow 0$
- $\{w_{ij}, \forall i,j\} \leftarrow \infty$
- $n \leftarrow 0$
- $C_p \leftarrow$ some constant > 0
  - For this example, $C_p = \frac{1}{2\sqrt{2}}$
- $\bar{X}_j \leftarrow 0$
  - Mean value/reward of selecting this position

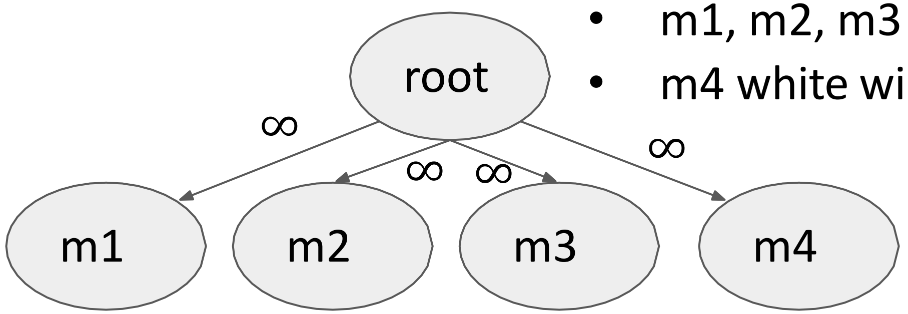
$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

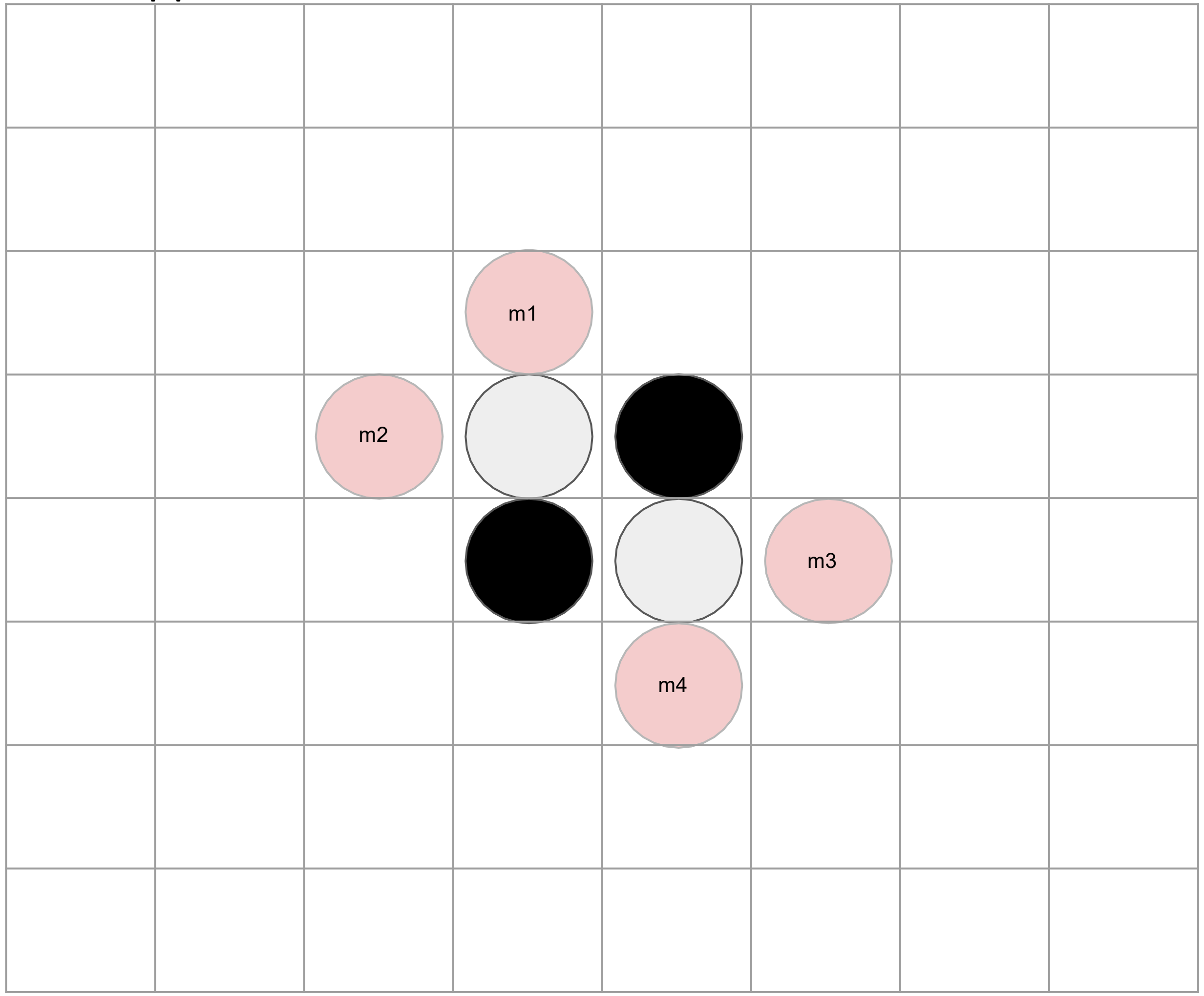
Example - The Game of Othello cont.

After first 4 iterations of simulation, suppose...

- m1, m2, m3 black wins
- m4 white wins



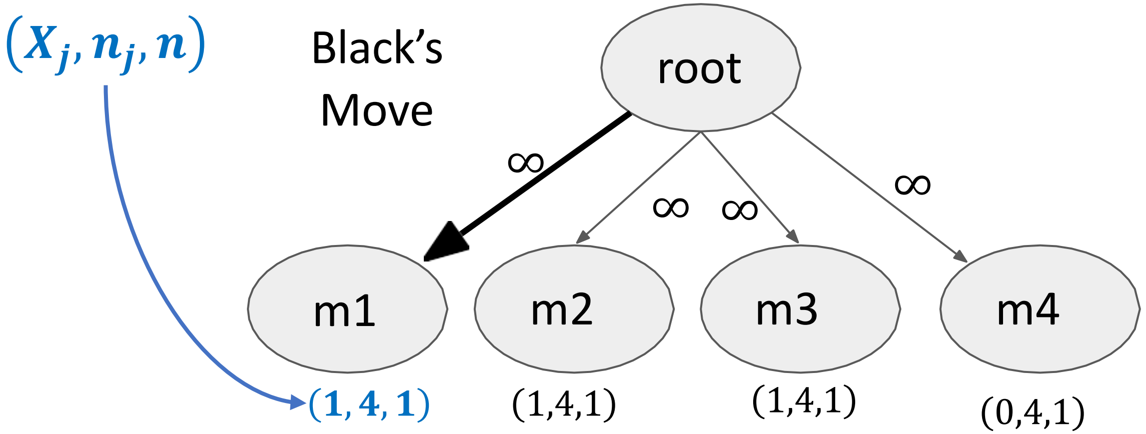
	X_j	n	n_j
m1	1	4	1
m2	1	4	1
m3	1	4	1
m4	0	4	1

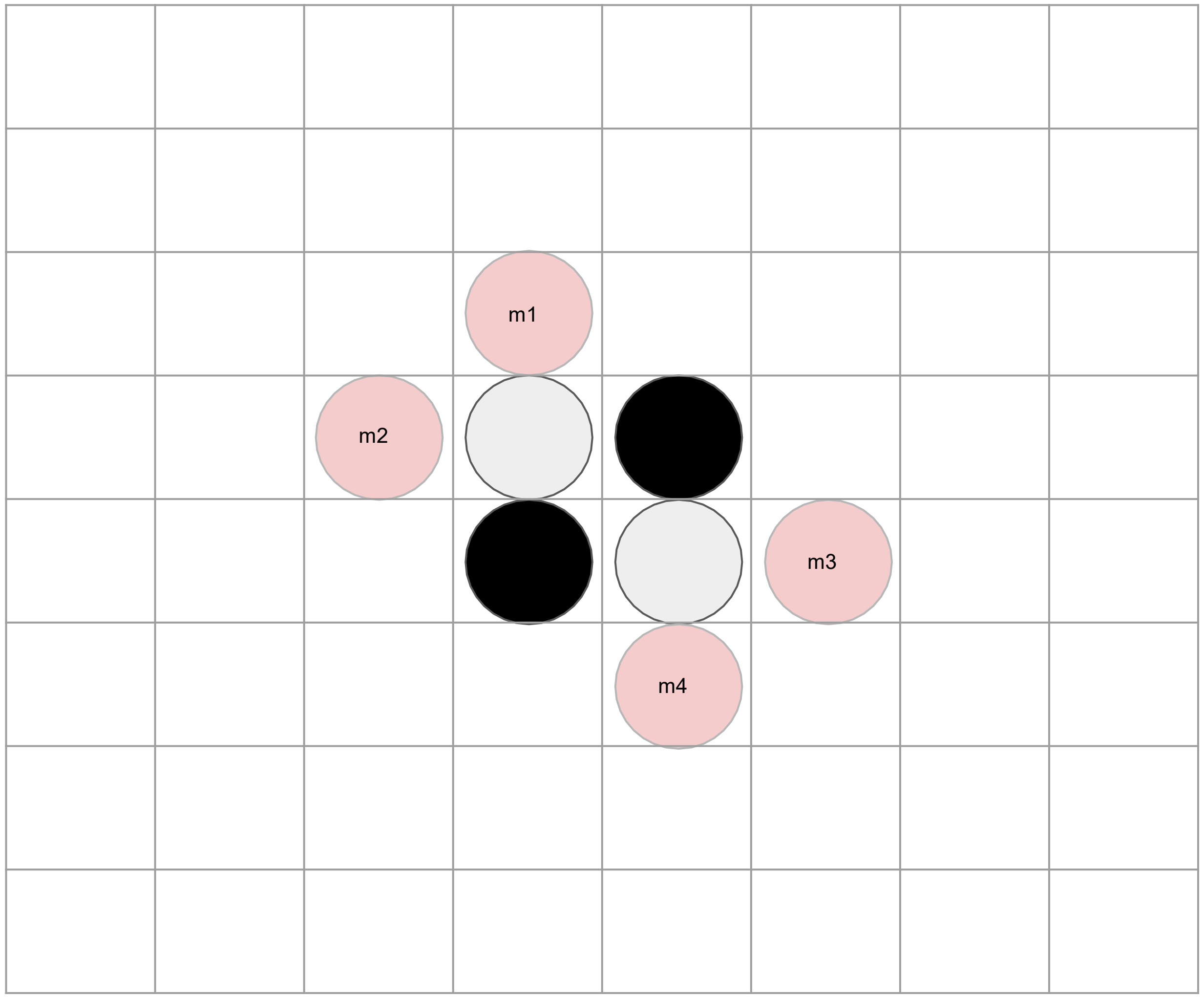


$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello cont.





After first 4 iterations of simulation, suppose…

- m1, m2, m3 black wins
- m4 white wins

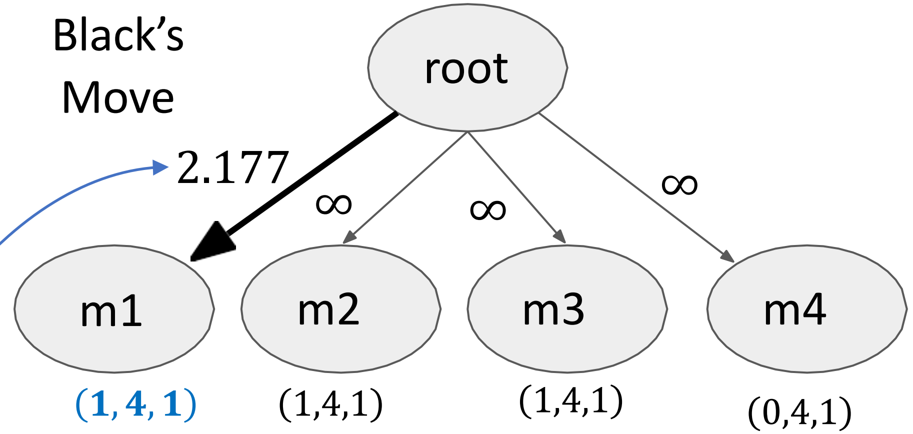
$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello cont.

(X_j, n_j, n)

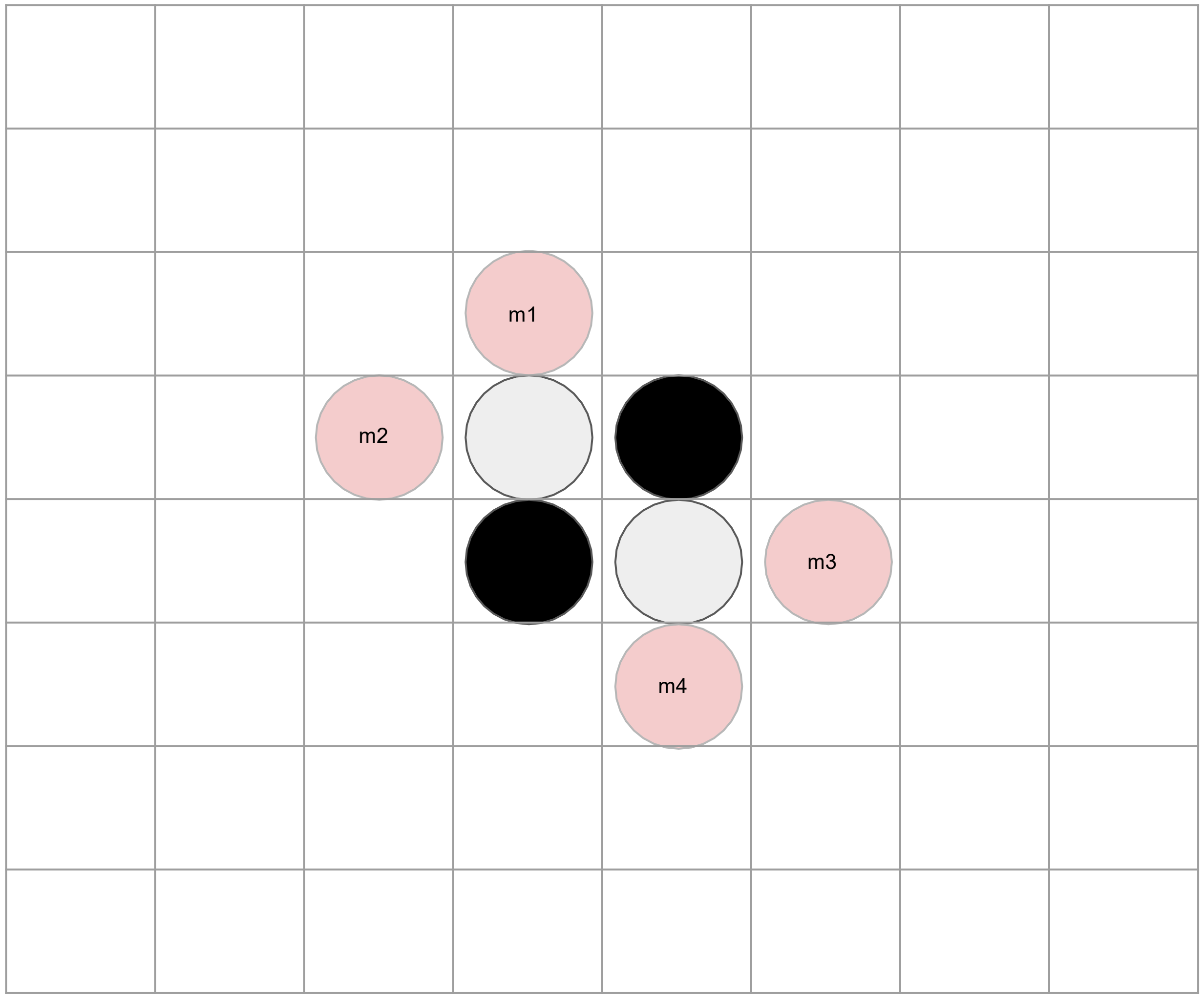
Black's Move



$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

$$= 1 + 2 * \left(\frac{1}{2\sqrt{2}}\right)\sqrt{\frac{2\ln 4}{1}}$$

$$w_{root,m1} = 2.177$$



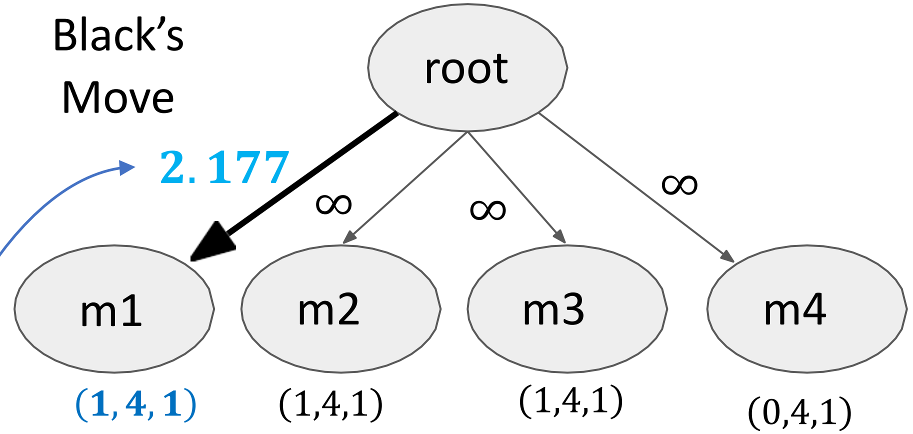
$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello cont.

$(\boldsymbol{X_j, n_j, n})$

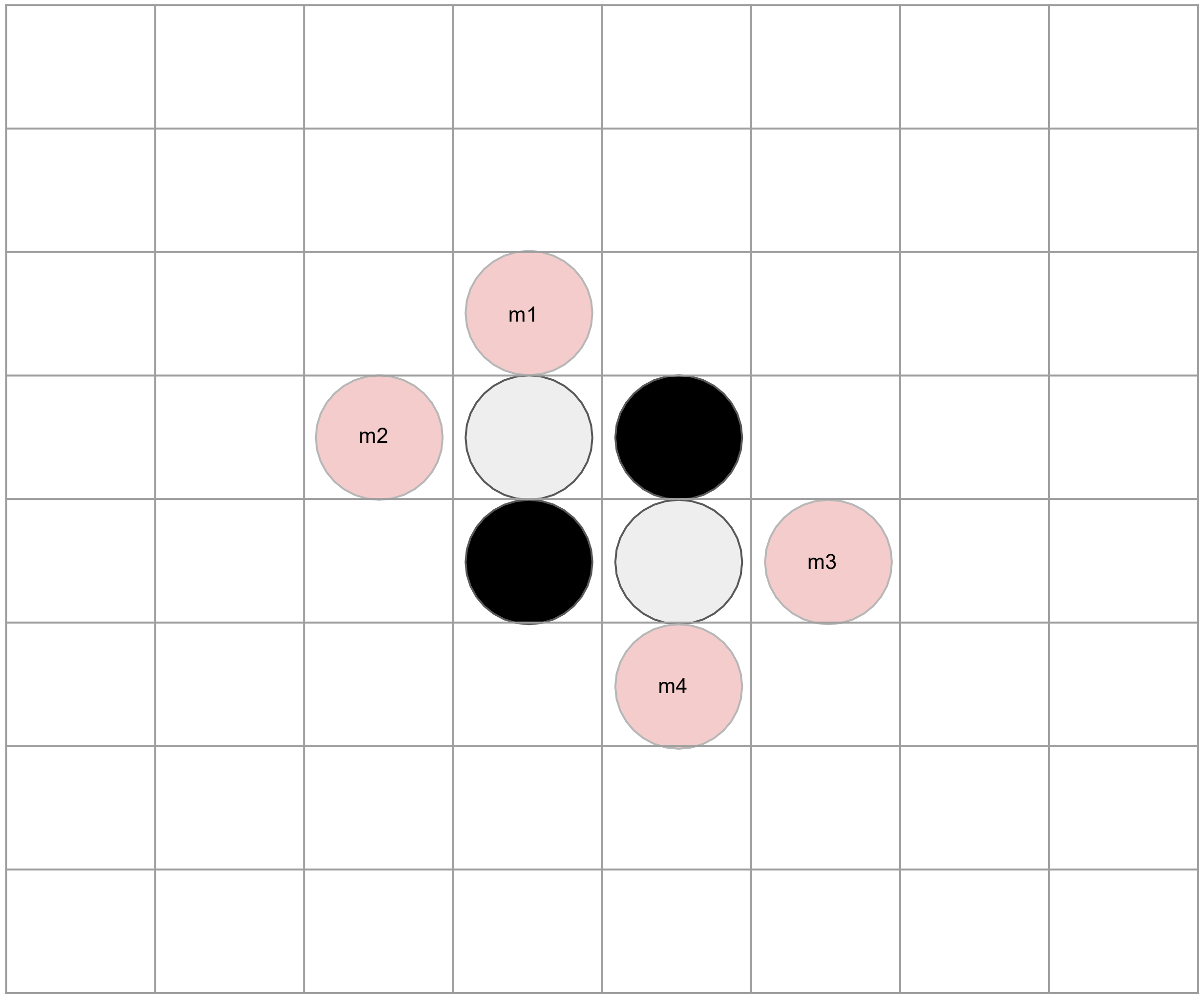
Black's Move



$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

$$= 1 + 2 * \left(\frac{1}{2\sqrt{2}}\right)\sqrt{\frac{2\ln 4}{1}}$$

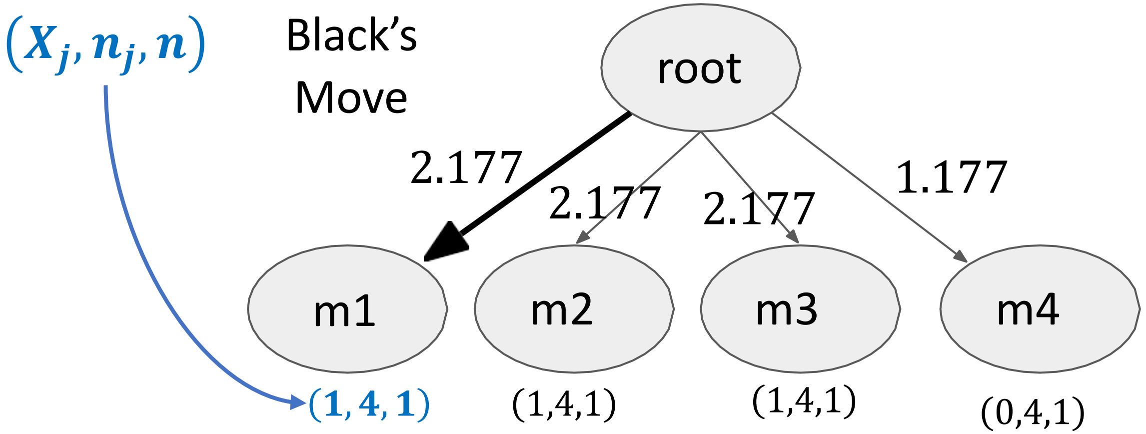
$$w_{root,m1} = 2.177$$

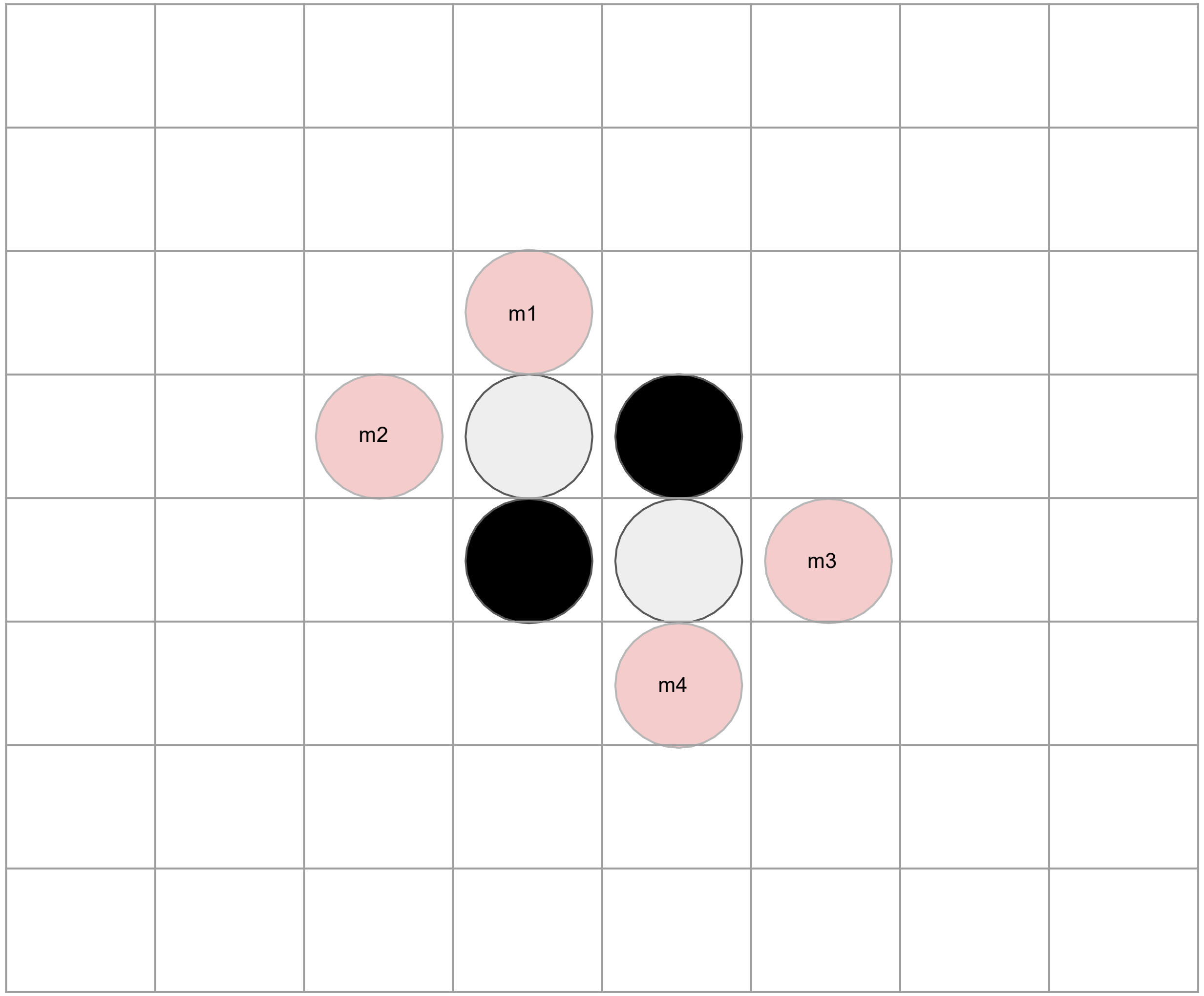


$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello cont.





$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

$$= 1 + 2 * \left(\frac{1}{2\sqrt{2}}\right)\sqrt{\frac{2\ln 4}{1}}$$

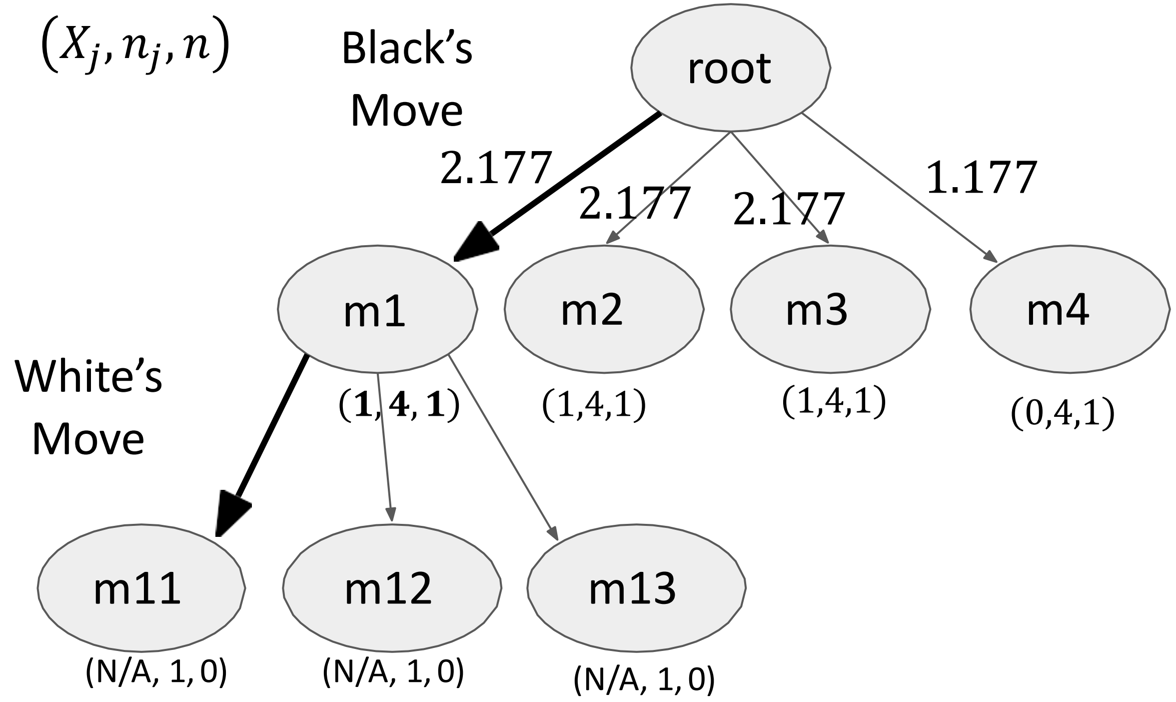
$$w_{root,m1} = 2.177$$

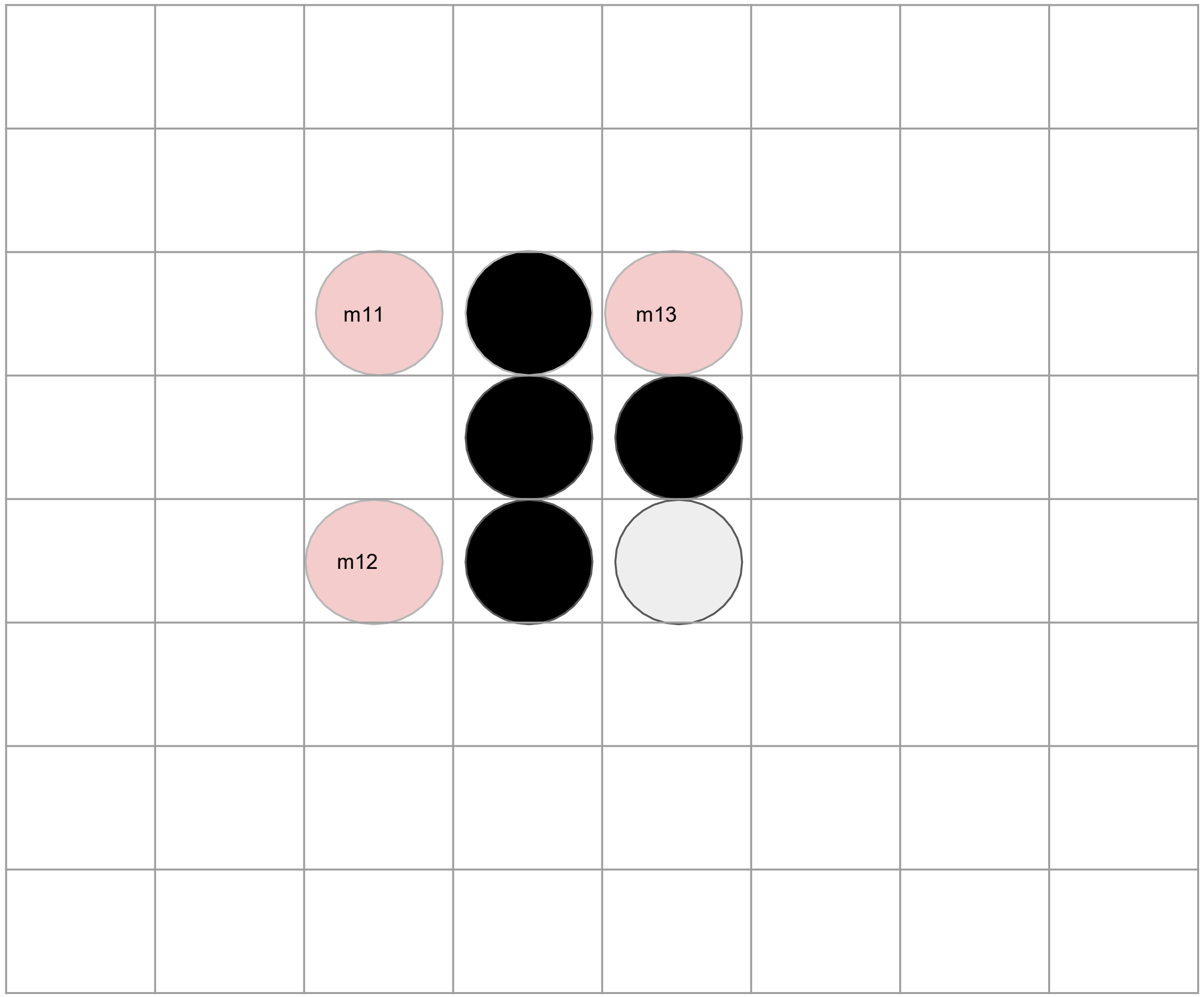
$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello Iter #5

(X_j, n_j, n)





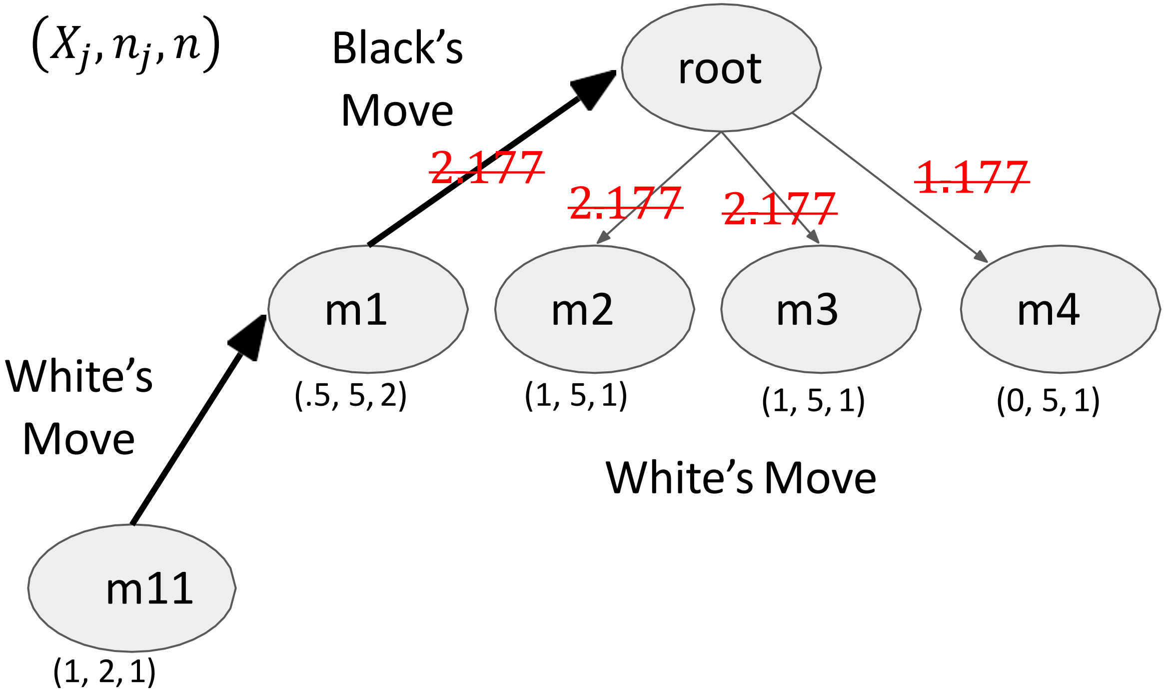
- First (black) selection picks m1
- Second (white) selection picks m11

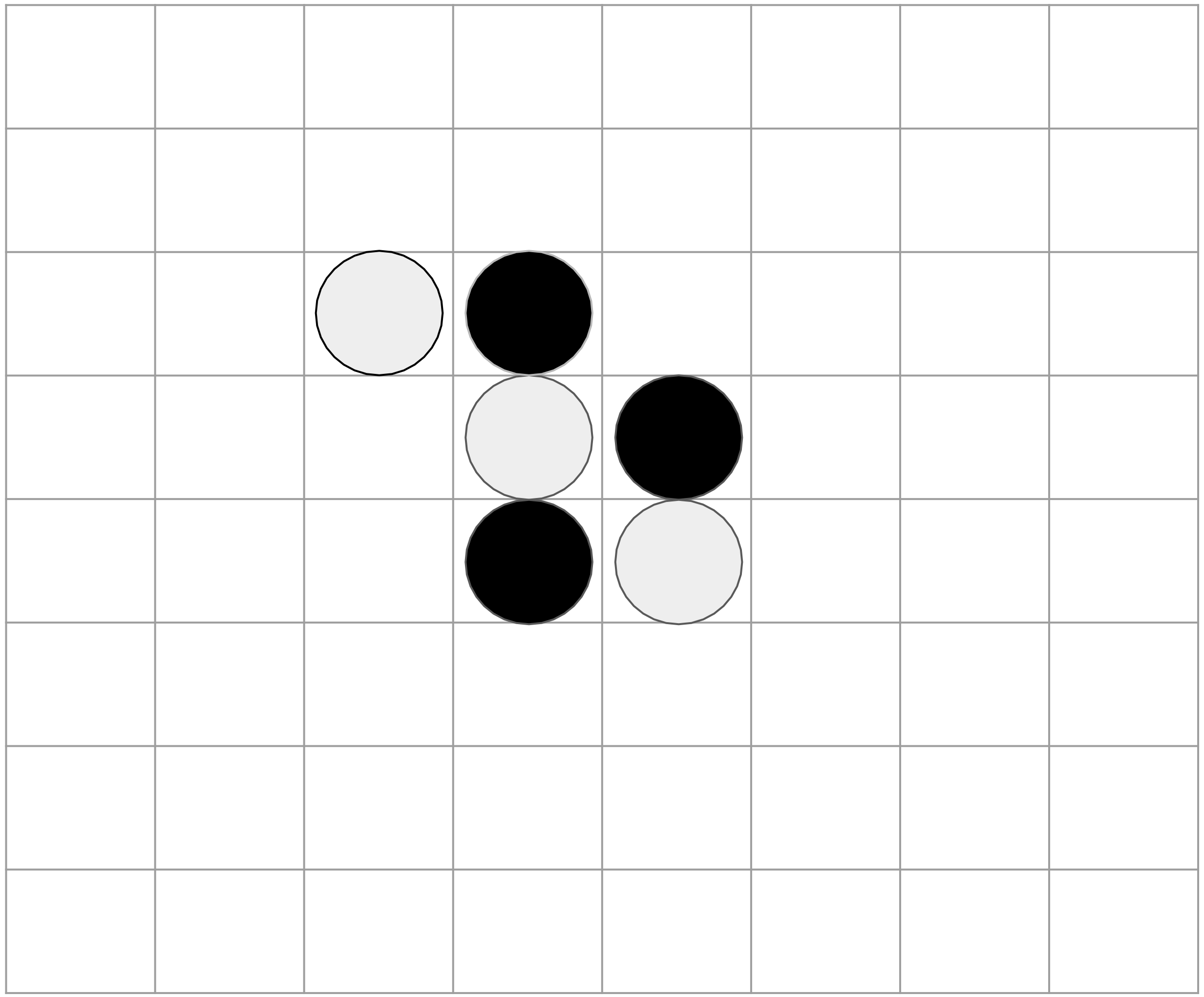
$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello Iter #5

(X_j, n_j, n)





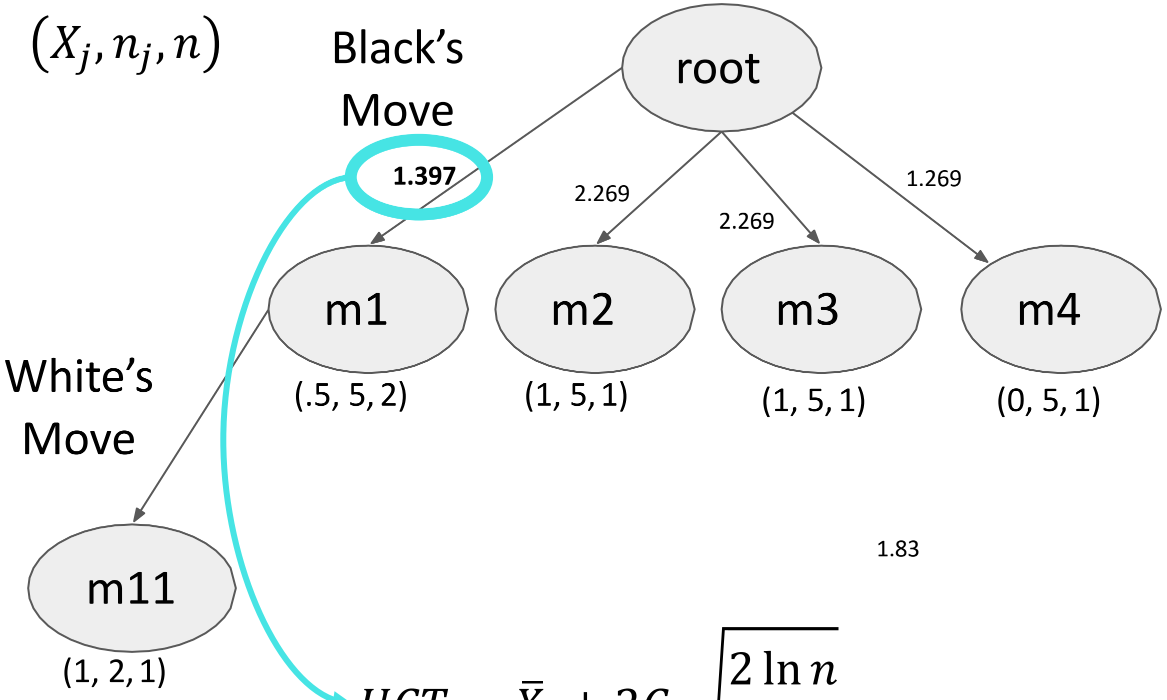
- Run a simulation
- White Wins
  - i.e., $\bar{X}_{m11} = 1$
- Backtrack, and update mean scores accordingly.

$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello Iter #6

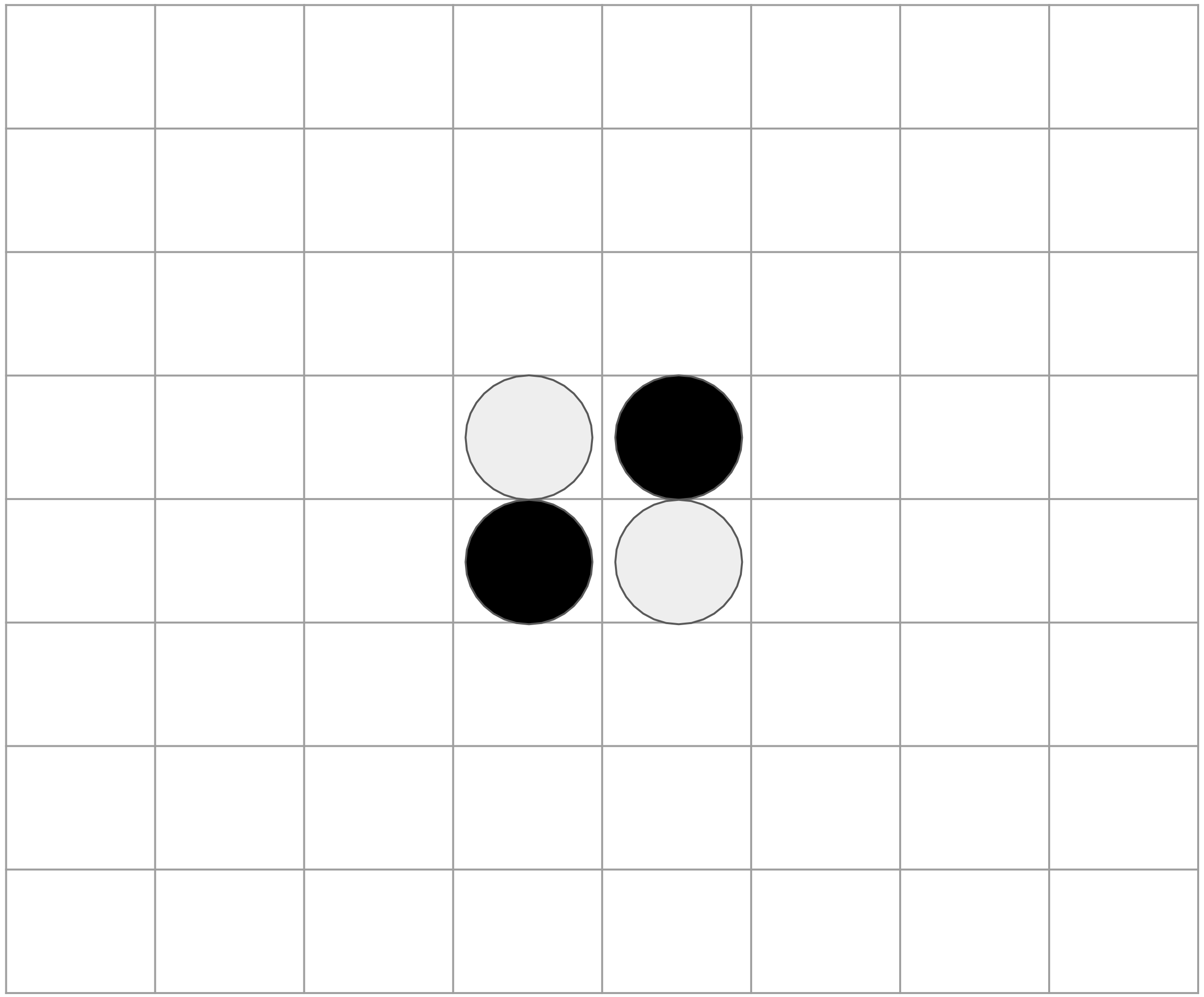
(X_j, n_j, n)



$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

$$= .5 + 2 * \left(\frac{1}{2\sqrt{2}}\right)\sqrt{\frac{2\ln 5}{2}}$$

$$= 1.397$$

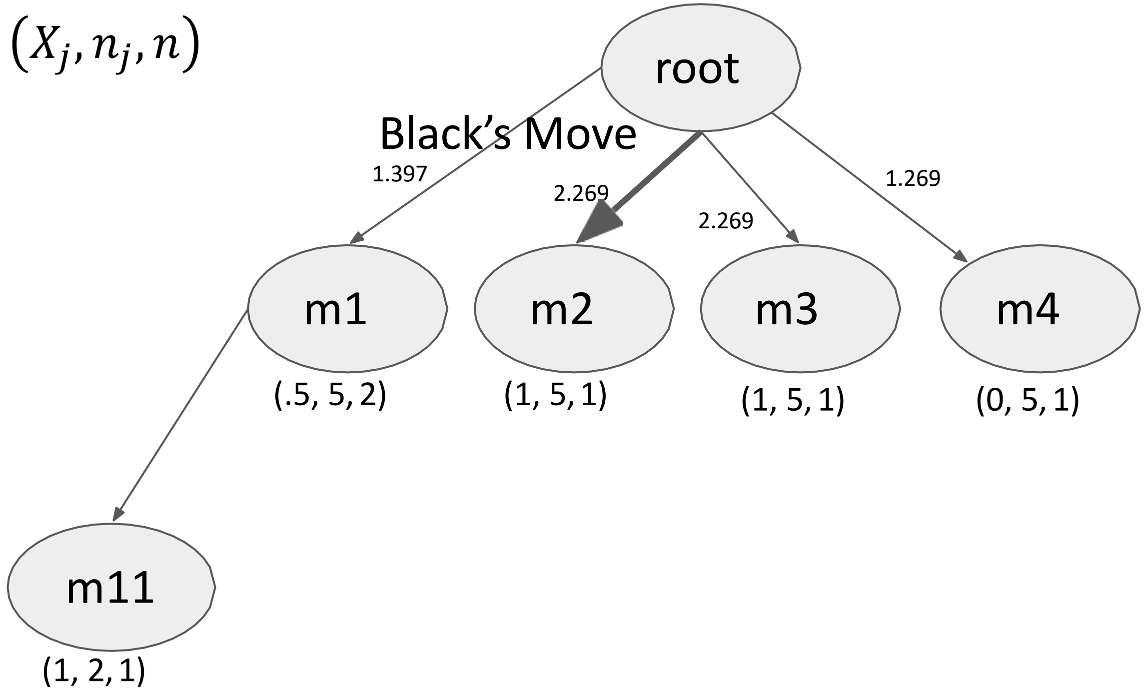


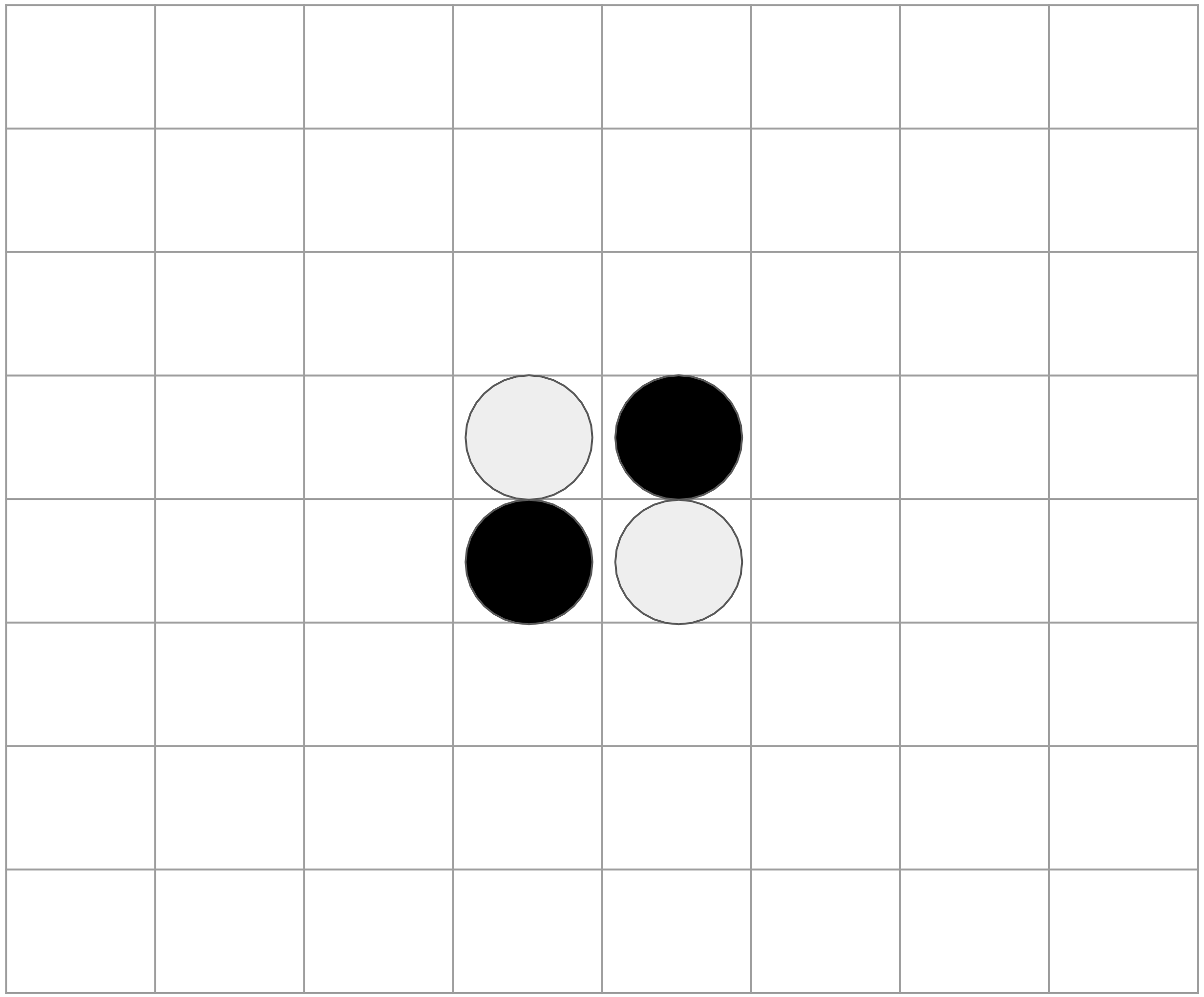
$$UCT = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

X_j – Mean value
n – # of parent visits
n_j – # of child visits

Example - The Game of Othello Iter #6

(X_j, n_j, n)



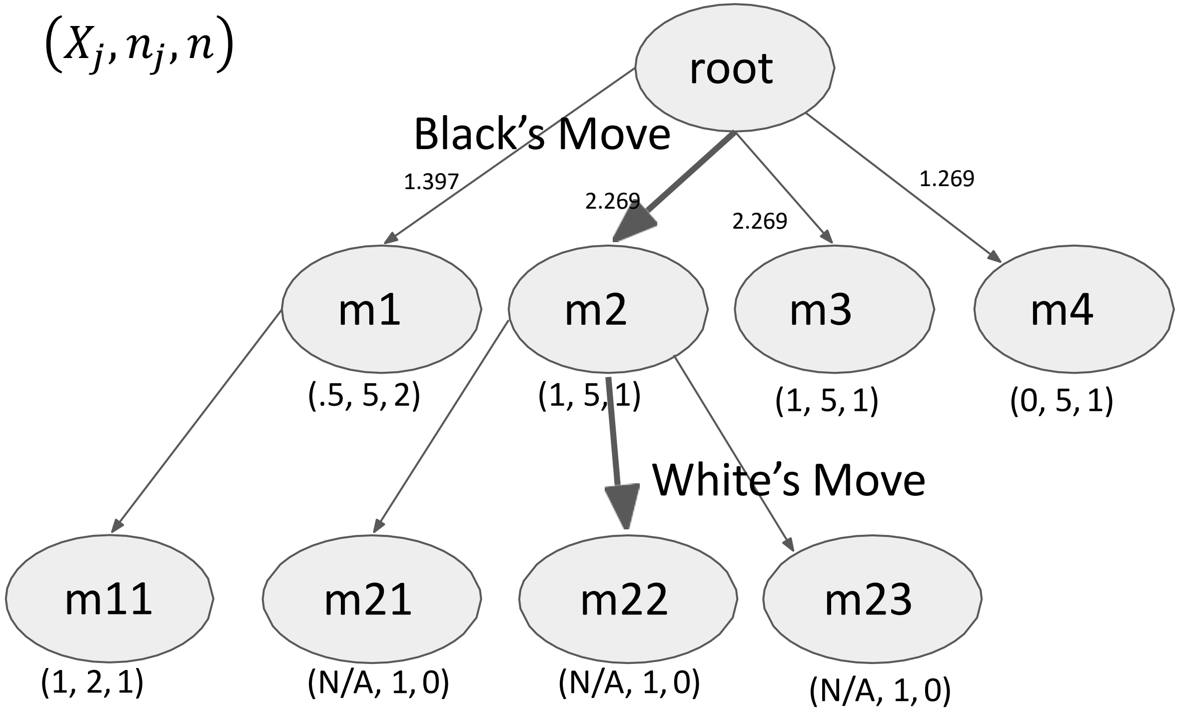


Suppose we first select m2 instead of m2

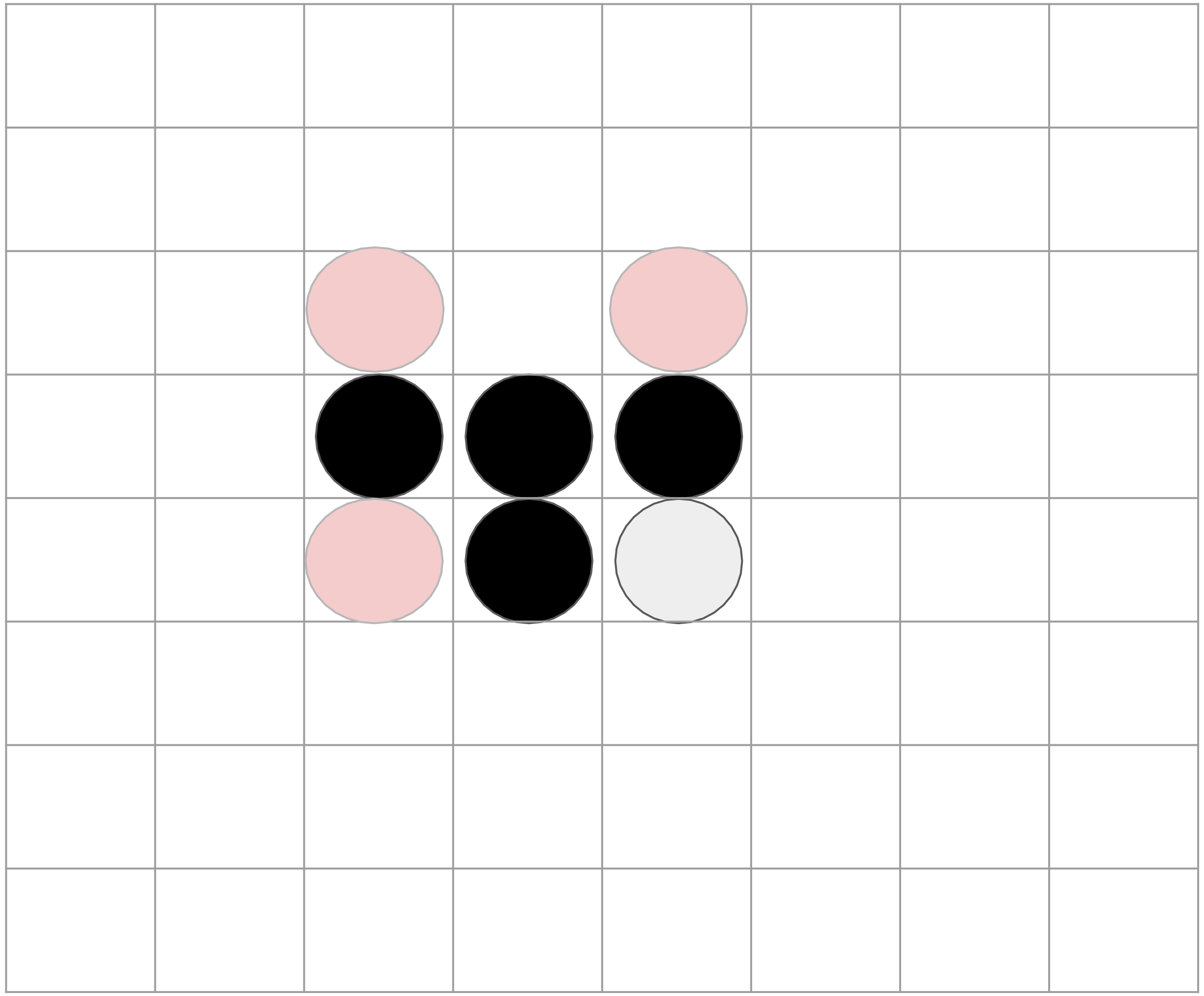
Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n_j, n)



(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



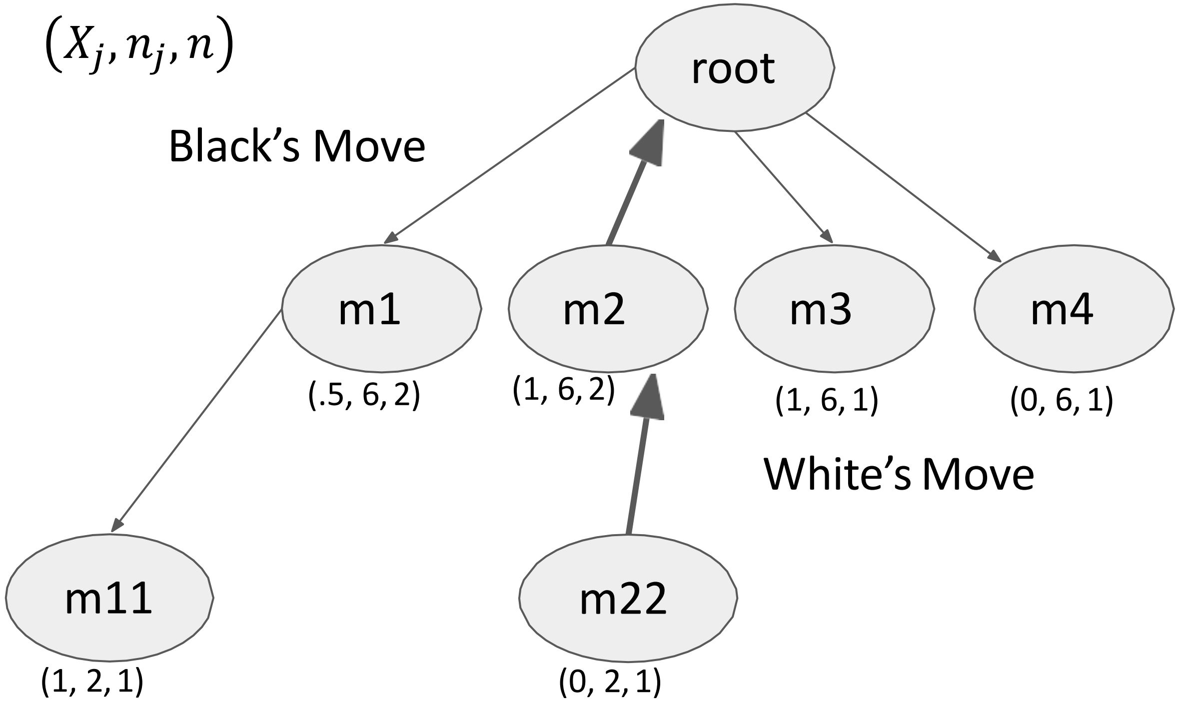
Suppose we then select m22

Note: Often, white is represented as a stochastic policy, and we sample white's move from that policy

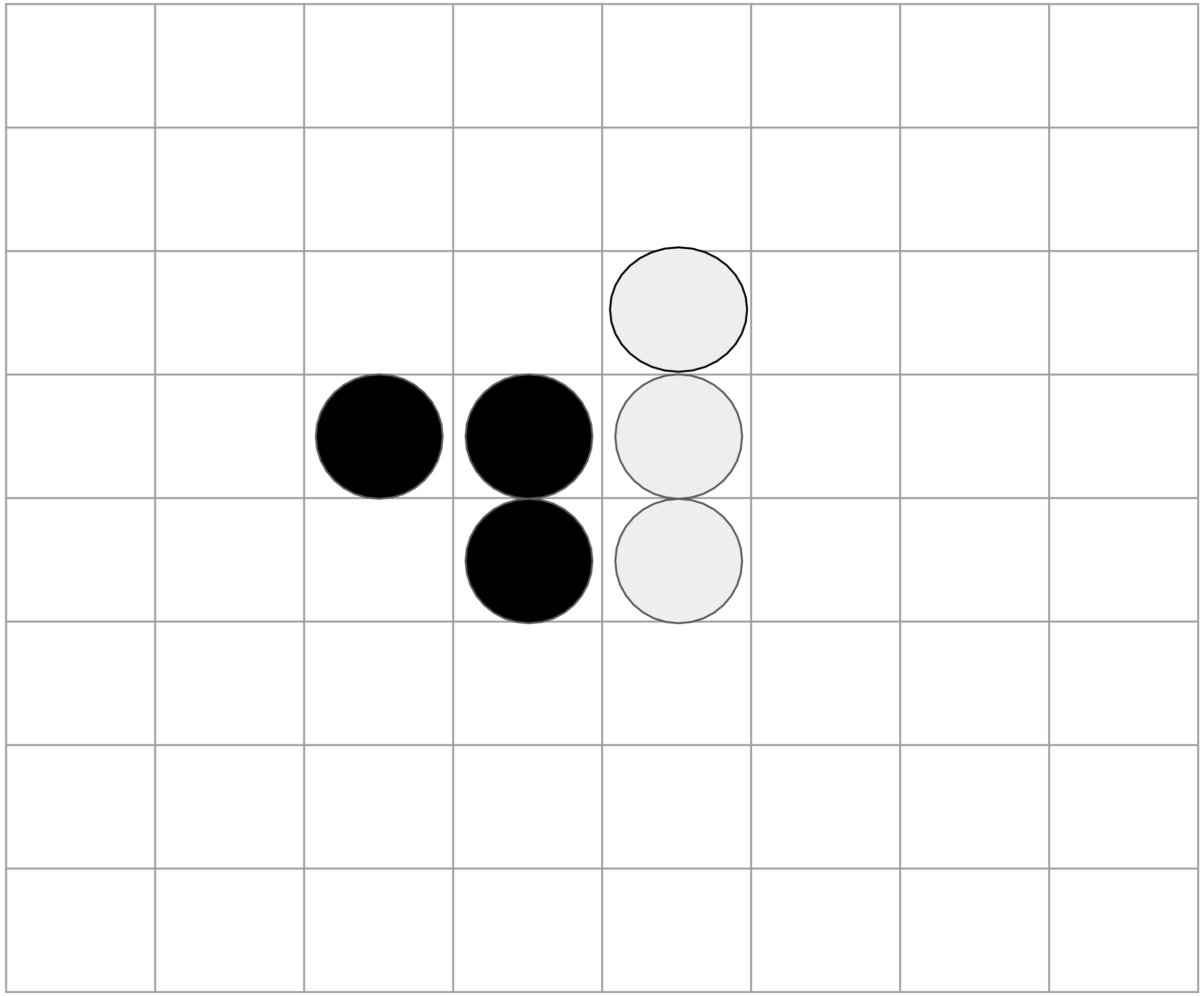
Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n_j, n)



(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



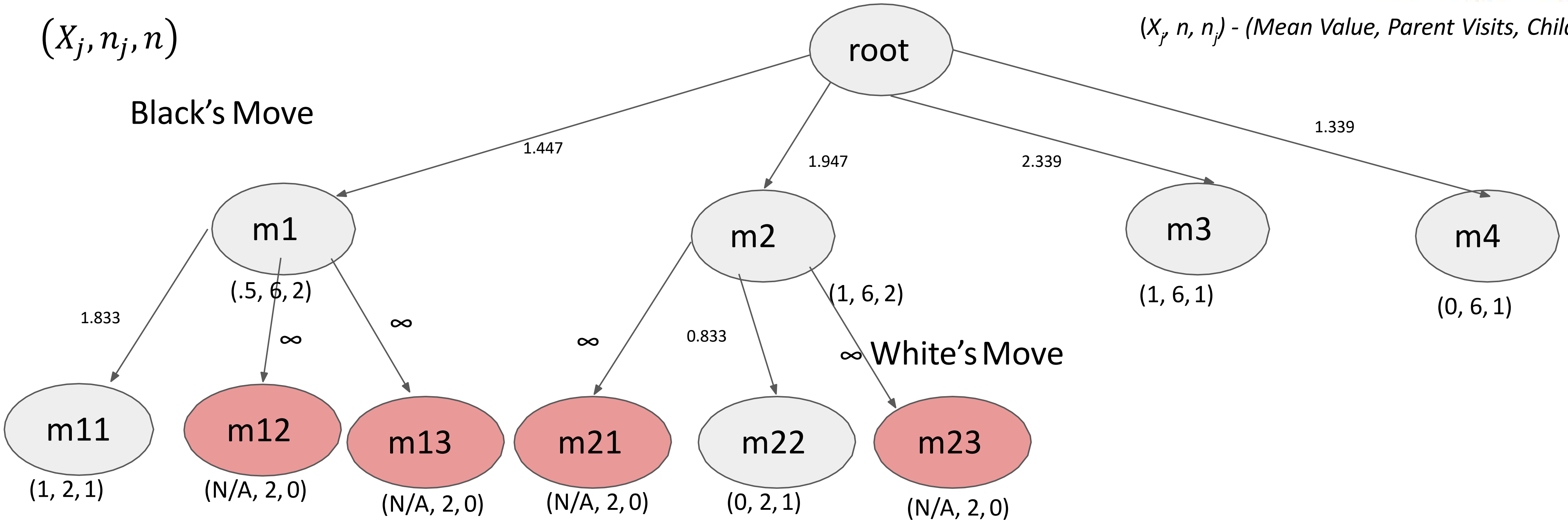
- Run simulated game from this position.
- Suppose black wins the simulated game
  - i.e., $\bar{X}_{m22} = 0$
- Backtrack and update values

Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n_j, n)

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



- This is how our tree looks after 6 iterations.
- Red Nodes not actually in tree
- Now given a tree, actual moves can be made using max, robust, max-robust, or other child selection policies.
- Only care about subtree after moves have been made

MCTS - Algorithm Recap

- Applied to solve Multi-Arm Bandit problem in a tree structure
    - UCT = UCB1 applied at each subproblem
- Due to tree structure same move can have different rewards in different subtrees
- Weight to go to a given node:
    - Mean value for paths involving node
    - Visits to node
    - Visits to parent node
    - Constant balancing exploration vs exploitation
- Determines values from Default Policy
- Determines how to choose child from Tree Policy
- Once you have acomplete tree - number of ways to pick moves during game - Max, Robust, Max-Robust, etc.

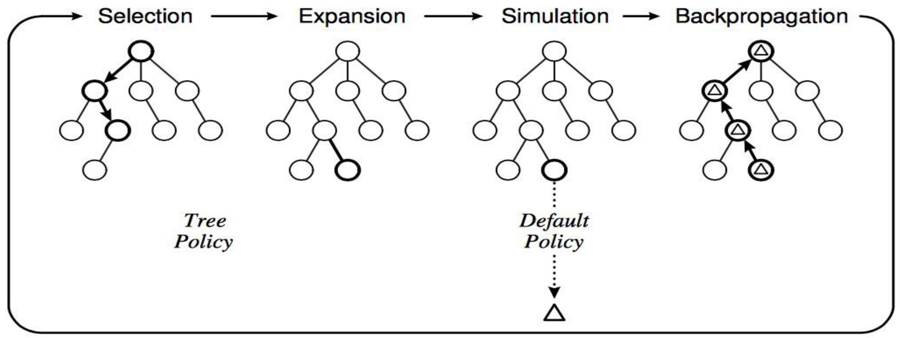


Fig. 2. One iteration of the general MCTS approach.

MCTS Case Study: AlphaGo

Go

- 2 player
- Zero-sum
- 19x19 board
- Very large search tree
    - Breadth ≈ 250, depth ≈ 150
    - Unlike chess
- No good heuristics
    - Human intuition hard to replicate
- Great candidate for applying MCTS
    - Vanilla MCTS not good enough

MCTS Case Study: AlphaGo

Idea 1) Use supervised/reinforcement to learn a simulation policy

→ Intelligently play game to completion to get a sense of whether the outcome will be favorable

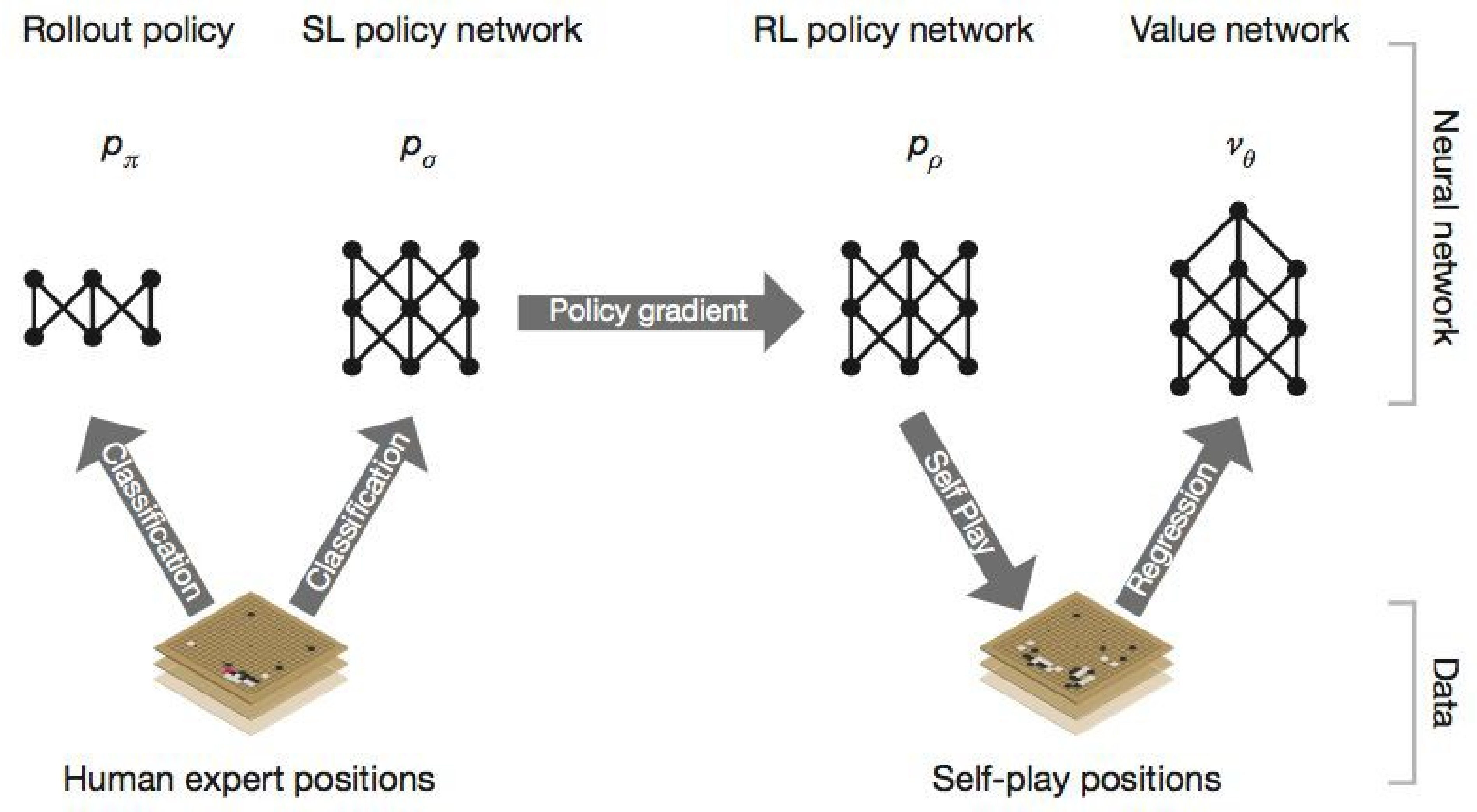
Idea 2) Use statistical inference (i.e., q-learning) to learn a cost-to-go-function (i.e., a value function) that can augment the simulation policy

→ Estimate the outcome of playing rest of game without actually playing rest of game

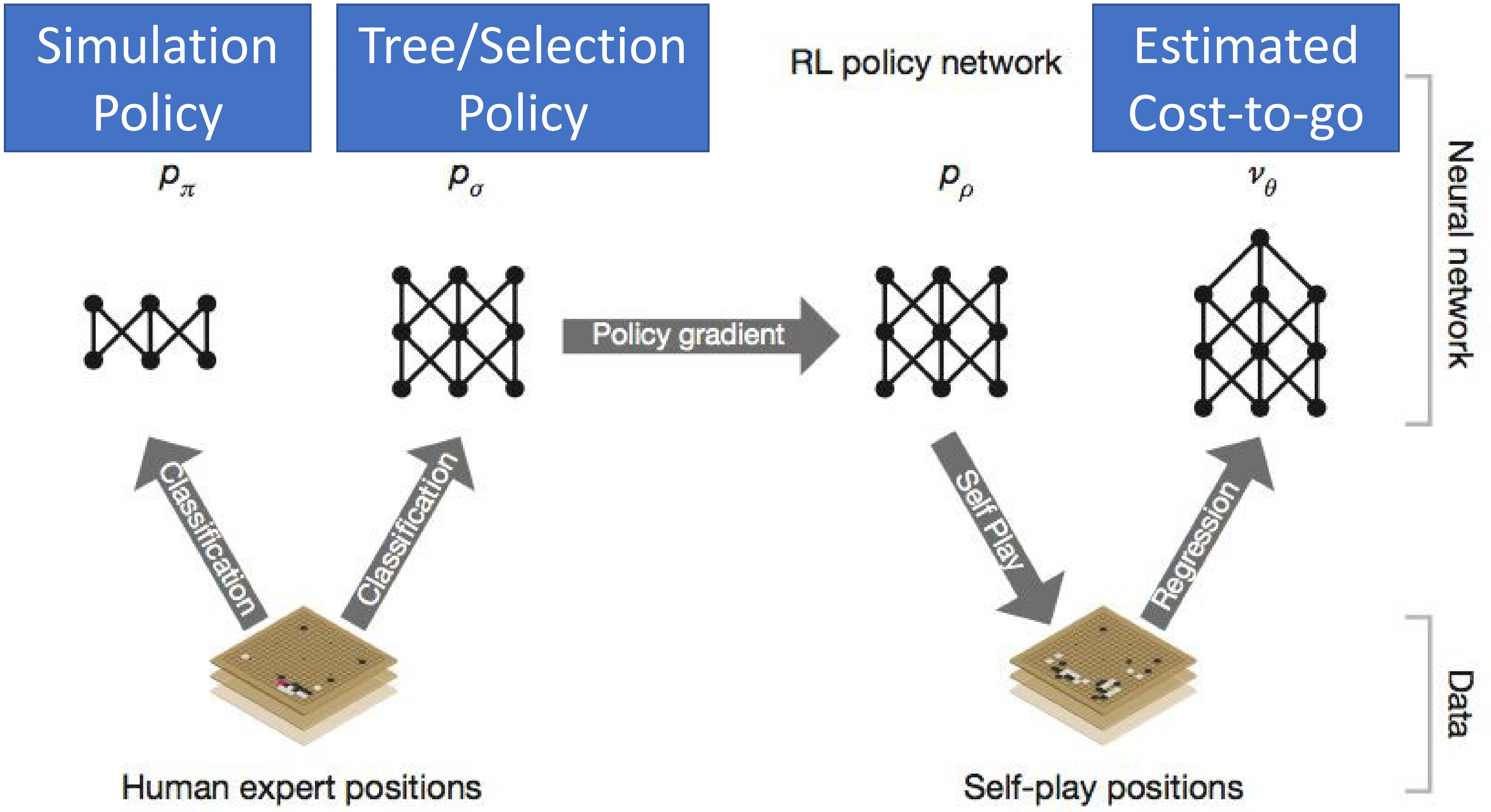
Idea 3) Use supervised/reinforcement learning to learn a tree policy

→ Try to make the most of each action opportunity

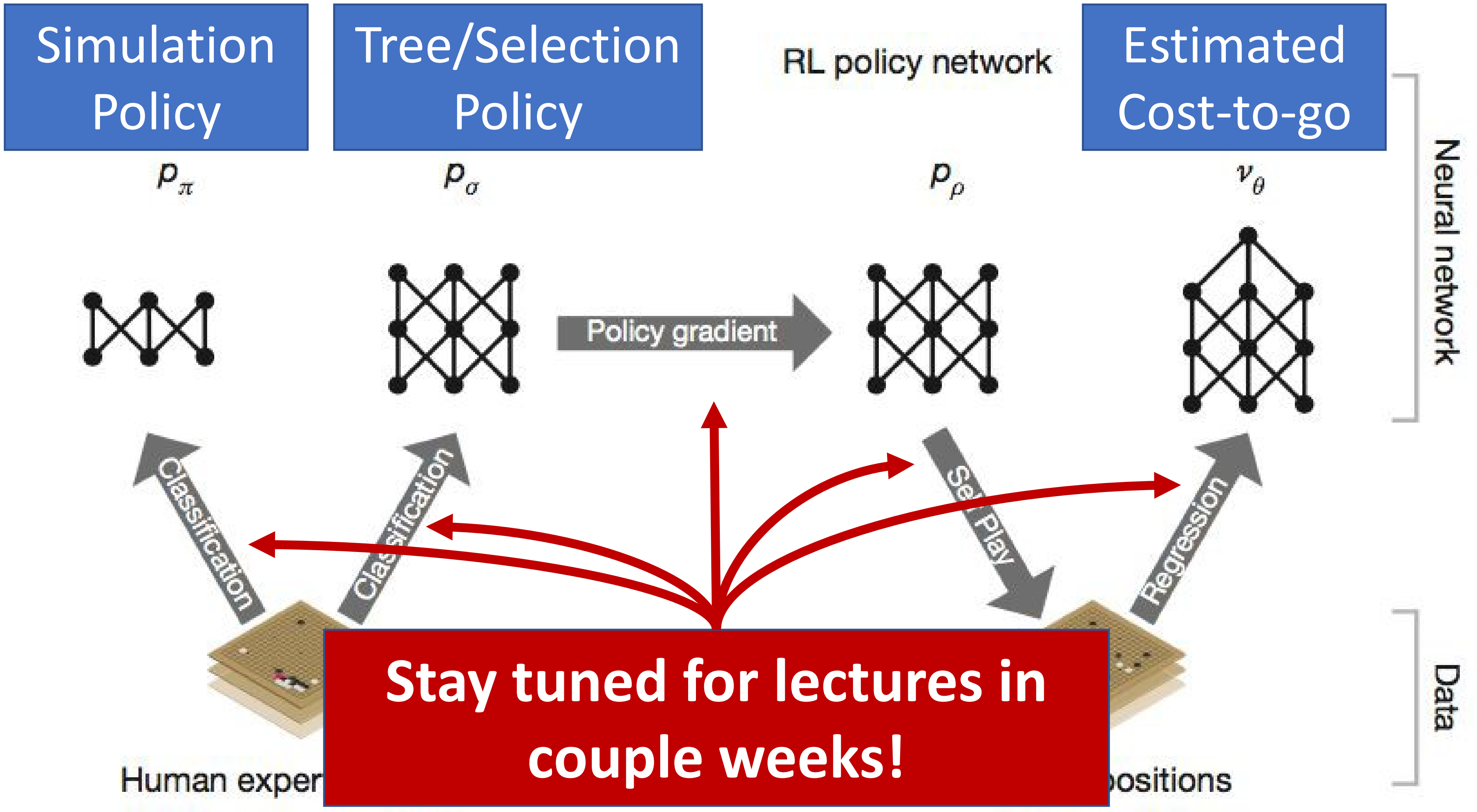
AlphaGo Architecture



AlphaGo Architecture



AlphaGo Architecture



AlphaGo: Selection/Tree Policy

$$a_t = \text{argmax}_a(Q(s_t, a) + u(s_t, a))$$

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

- a_t - action chosen for time step t given state s_t
- Q(s_t, a) – Average reward for playing a in state s_t (exploitation term)
  - **Not** the “Q-function” from q-learning, but rather an Alpha-Go specific term
- *P(s, a)* - prior expert probability of playing moving *a*
  - For AlphaGo, this was done via Supervised Learning (“Mimicry”)
- *N(s, a)* - number of times we have visited parent node
- $u(s_t, a)$ acts as a bonus value that decays with repeated visits

AlphaGo: Simulation/Value Policy

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z(s_L)$$

- The point estimate of the value of a node (representing state s) is given by the convex combination, controlled by λ, of two terms:
  - Cost-to-go-function, $v_\theta(s)$
    - Learned via reinforcement learning (technically regression)
  - Win/loss: The result from playing game to completion starting from state s using simulation policy z
    - This simulation policy is learned via supervised learning to “mimic” expert players.
    - The network is “small” to enable it to act quickly

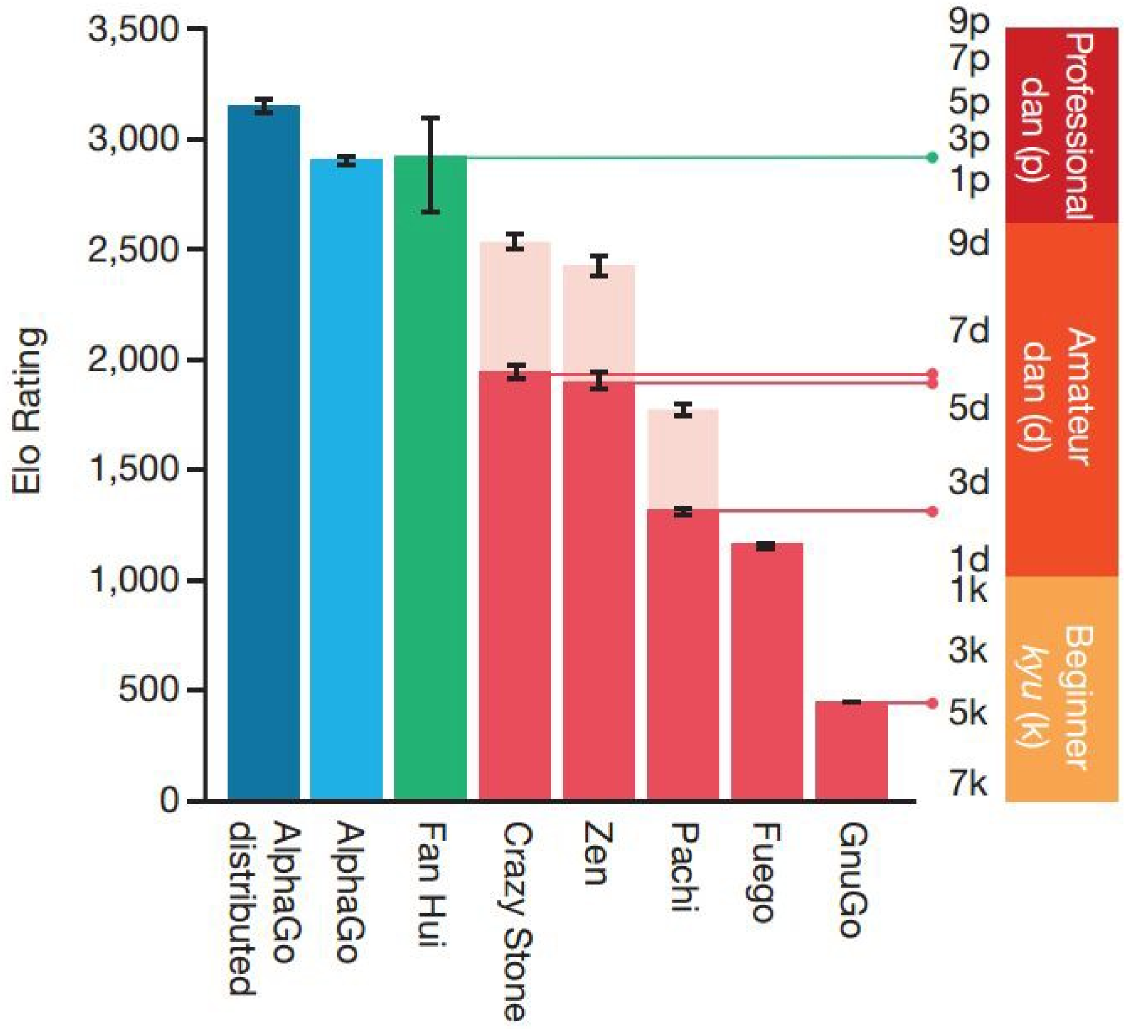
AlphaGo: Backpropagation

$$Q(s,a) = \frac{\sum_{i=1}^{n} 1_{(s,a,i)} V\left(s_L^i\right)}{\sum_{i=1}^{n} 1_{(s,a,i)}}$$

- Extra index i is to denote the i^{th} simulation with n total simulations
- Update visit count, mean reward of simulations passing through node

Once MCTS completes, the algorithm chooses the most-taken move from the root position

AlphaGo: Results



Takeaway

- A* search is an optimal, heuristic, search algorithm that works by relying on an admissible heuristic
  - Challenge: Good, admissible heuristics are hard to come by

  →In a few lectures, we will see how reinforcement learning can help!

- MCTS is key to modern (stochastic-)state-space search
  - Search heuristic provided (UCT: Apply UCB1 at each step)
  - Caveat: Need model of the opponent

- Key difference:
  - We are moving from a deterministic world (BFS, DFS, IDS, A, A*) vs. non-deterministic and random worlds

- Path Forward:
  - Next 1/3 of semester, we will build to understand how machine learning can help us learn "tree/simulation policies" and "value functions" in non-deterministic and random worlds!