CS 540 Introduction to Artificial Intelligence

Review (Topics after midterm)

University of Wisconsin-Madison
Spring 2026 Sections 1 & 2

Announcements: Final Information

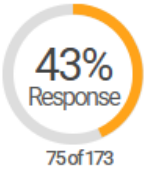
- **Time: May 7th, 12:25 - 2:25 PM**
- **Location for sections 001 AND 002 (Instructor Blerina Gkotse) based on your last name:**
  - Ingraham B10 (Last name starting with A-P inclusive)
  - Noland 132 (Last name starting with Q-Z inclusive)
- Location for **section 003 (Instructor Young Wu)**: Sterling 1310
- Students with McBurney accommodations or alternate requests should have received an email with additional information.

Announcements: Final Information

- **Topics**: The exam is **cumulative**
- Exclusion list (Sections 1&2):
  - Logic
  - Tragedy of the Commons (Games I)
  - “From Extensive Form to Normal Form” (Games II)
- **Practice questions**: Canvas -> Files -> Past Exams
- **Format**: MCQ.
- **Cheat Sheet**: up to 2 handwritten sheets, front and back
- **Calculator**: Calculators are allowed if they don't have an internet connection. A calculator will not be necessary though it may be useful to double check simple arithmetic.
- **Bring**: your WISC ID, pencil (No 2 or softer) and your cheat sheets.

Announcements

- **Course evaluation until May 1st:**
  - If participation reaches 50%, we'll reveal more information for the final.
  - If participation reaches 80%, even more.

Section 001 

Section 002 

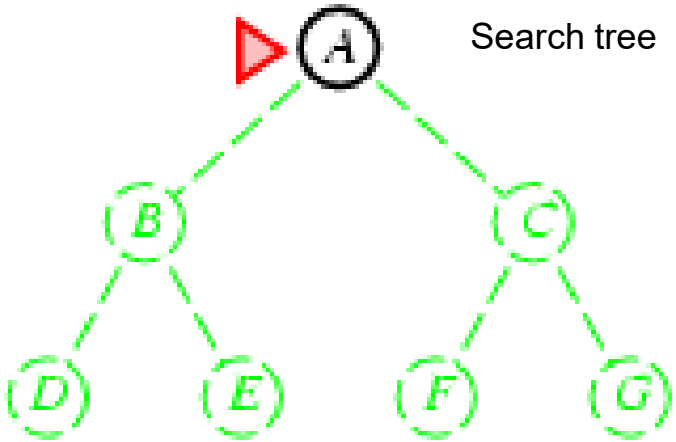
- Today: Last lecture**, review topics covered after the midterm**.
- Please note, exam is **cumulative**.
- For earlier topics, please watch the recordings of the review lectures before the midterm.

Uninformed Search

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



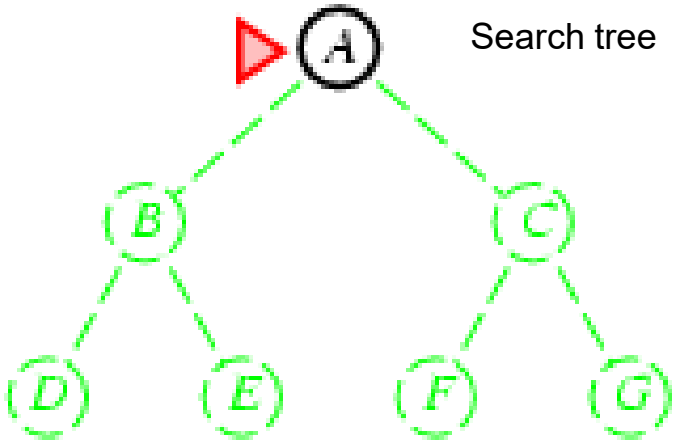
Search tree

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

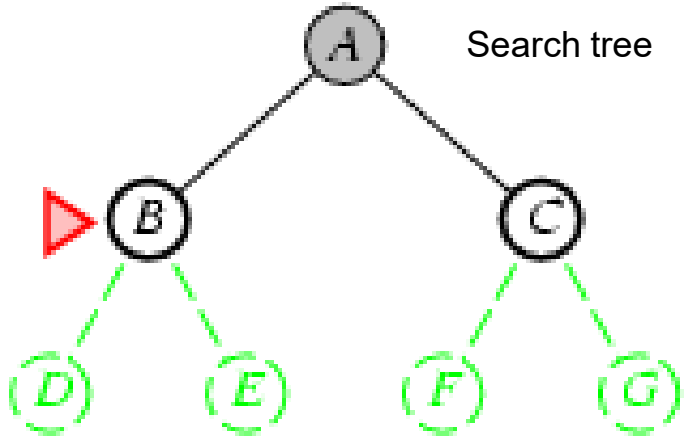
queue (fringe, OPEN)
→ [A] →

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

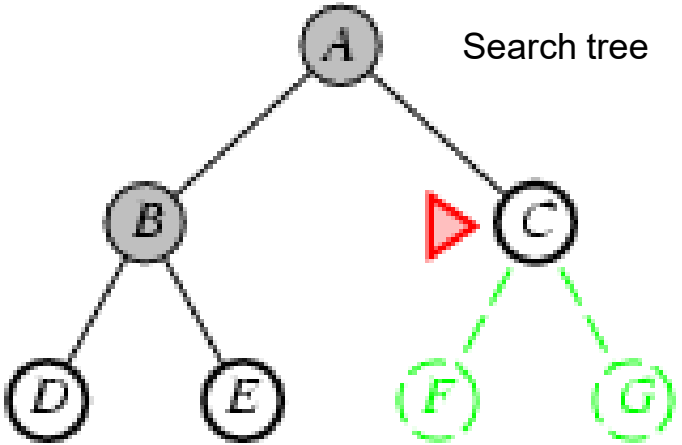
queue (fringe, OPEN)
→ [CB] → A

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

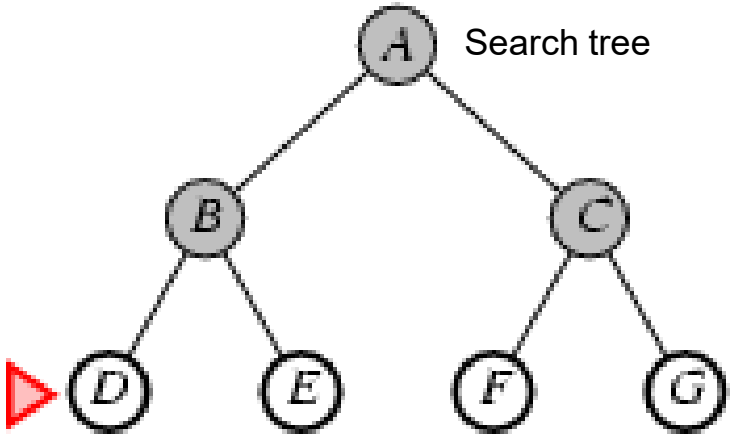
queue (fringe, OPEN)
→ [EDC] → B

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue (fringe, OPEN)
□[GFED] → C

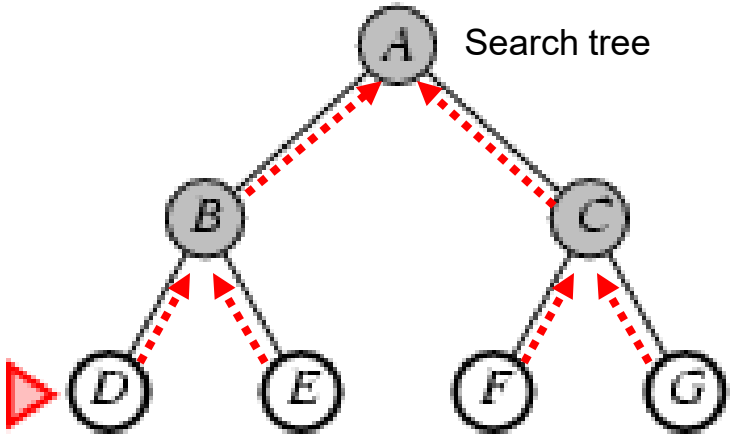
If G is a goal, we've seen it, but we don't stop!

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue
□[] →G

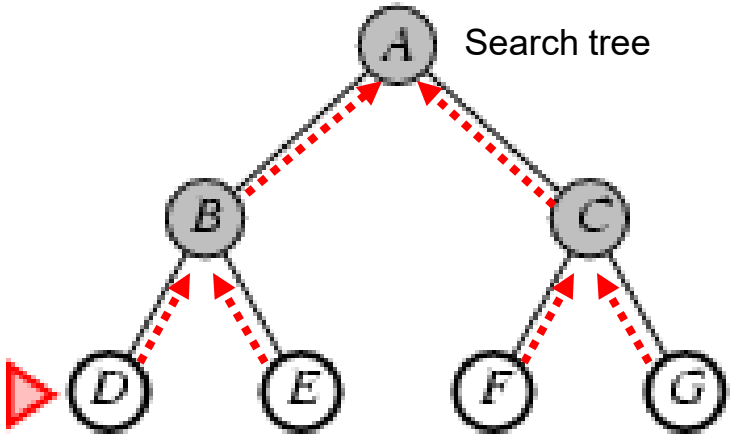
... until much later we pop G.

Looking foolish? Indeed. But let's be consistent…

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

queue
□[] →G

... until much later we pop G.

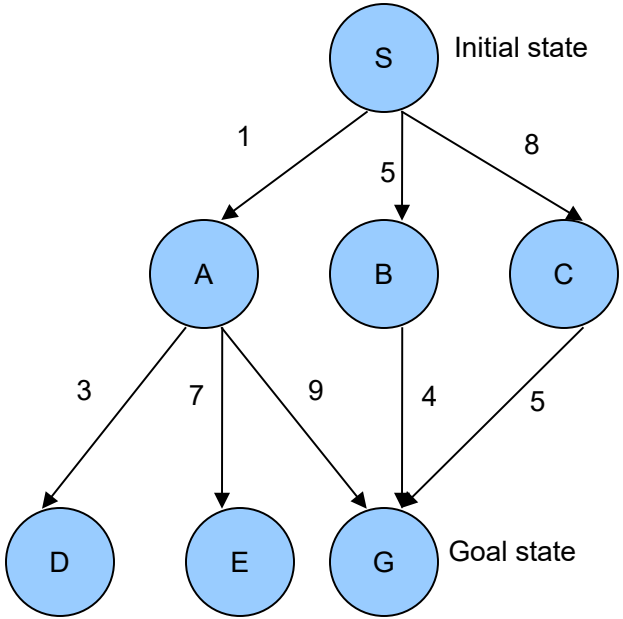
We need back pointers to recover the solution path.

Looking foolish? Indeed. But let's be consistent…

Uniform-cost search

- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path).
- Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item

Example



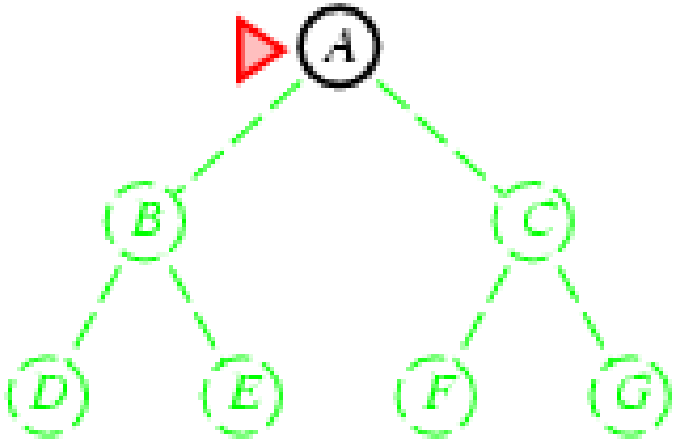
1: (S,0), [(A,1), (B,5), (C,8)]
2: (A,1), [(B,5), (C,8), (D,4), (E,8), (G,10)]
3: (D,4), [(B,5), (C,8), (E,8), (G,10)]
4: (B,5), [(C,8), (E,8), (G,9)]
5: (C,8), [(E,8), (G,9)]
6: (E,8), [(G,9)]
7: (G,9), []: Success!

(All edges are directed, pointing downwards)

Depth-first search (DFS)

Use a stack (First-in Last-out)

1. push(Initial states)
2. While (stack not empty)
3. s = pop()
4. if (s==goal) success!
5. T = succs(s)
6. push(T)
7. endWhile



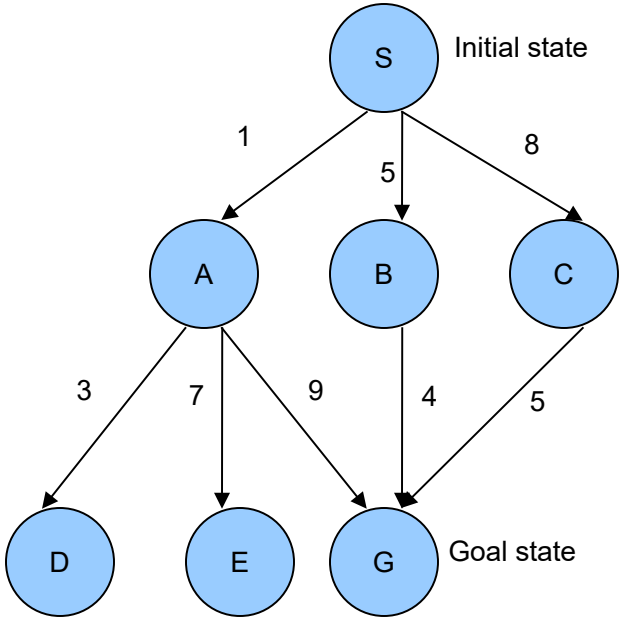
stack (fringe)

1. A, [B, C]
2. B, [D, E, C]
3. D, [E, C]
4. E, [C]
5. C, [F, G]
6. F, [G]
7. G

Iterative deepening

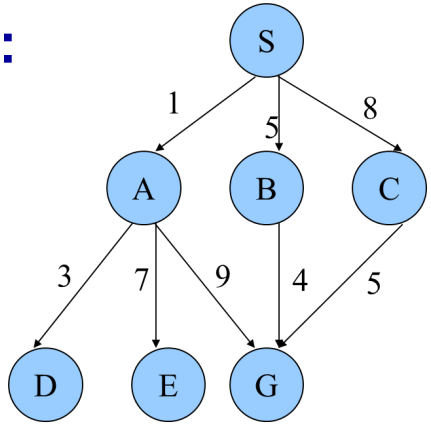
- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
  - Time complexity like BFS
- Preferred uninformed search method

Example



(All edges are directed, pointing downwards)

Nodes expanded by:



- Breadth-First Search: S A B C D E G

  Solution found: S A G

- Uniform-Cost Search: S A D B C E G

  Solution found: S B G (This is the only uninformed search that worries about costs.)

- Depth-First Search: S A D E G

  Solution found: S A G

- Iterative-Deepening Search: S A B C S A D E G

  Solution found: S A G

Performance of search algorithms on trees

b: branching factor (assume finite) d: goal depth m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	O(bm)
Iterative deepening	Y	Y, if [1]	$O(b^d)$	O(bd)

1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C* is the best goal path cost.

Informed Search

Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

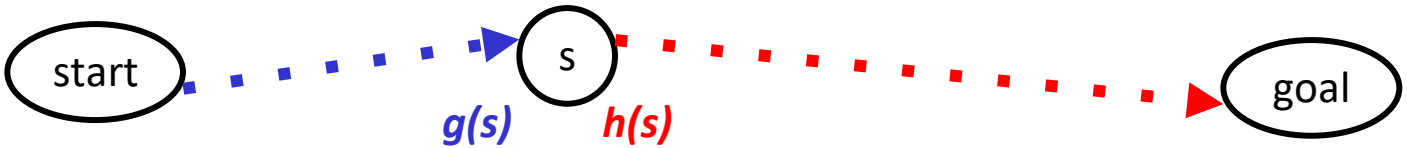
Path cost ***g***(*s*) from start to node *s*

Successors.

Informed search. Know:

All uninformed search properties, plus

Heuristic **h(s)** from s to goal (recall game heuristic)



A* Search

Use ***g*(*s*)** + ***h*(*s*)**, with one **requirement**

Demand that ***h*(*s*)** ≤ ***h**(*s*)** where **h*(s)** is true cost from s to goal.

If heuristic has this property, it is called “admissible”

- Optimistic! Never over-estimates

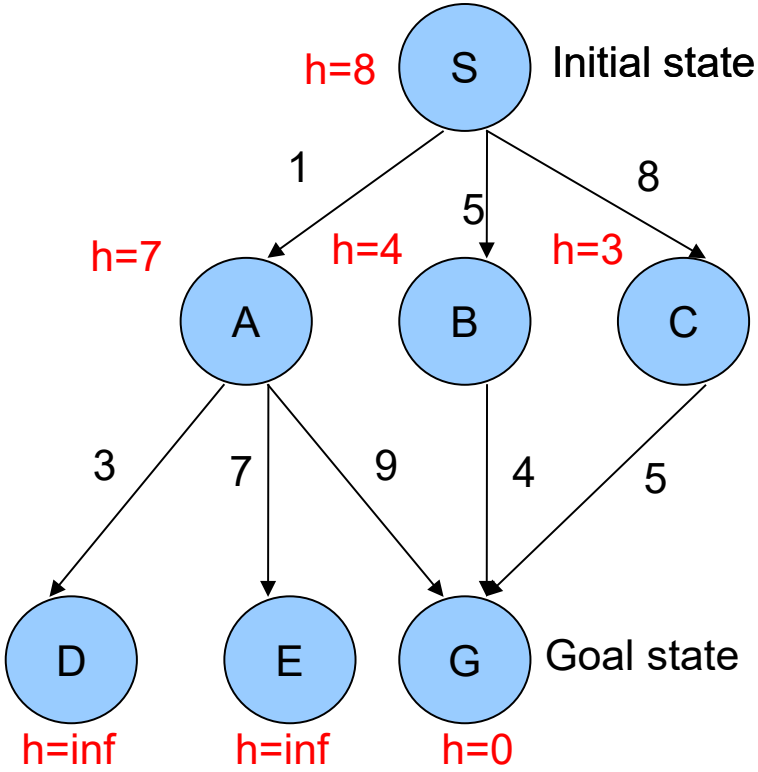
Still need ***h*(*s*)** ≥ 0

- Negative heuristics can lead to strange behavior

This is **A* search**

Recap and Examples

Example for A*:

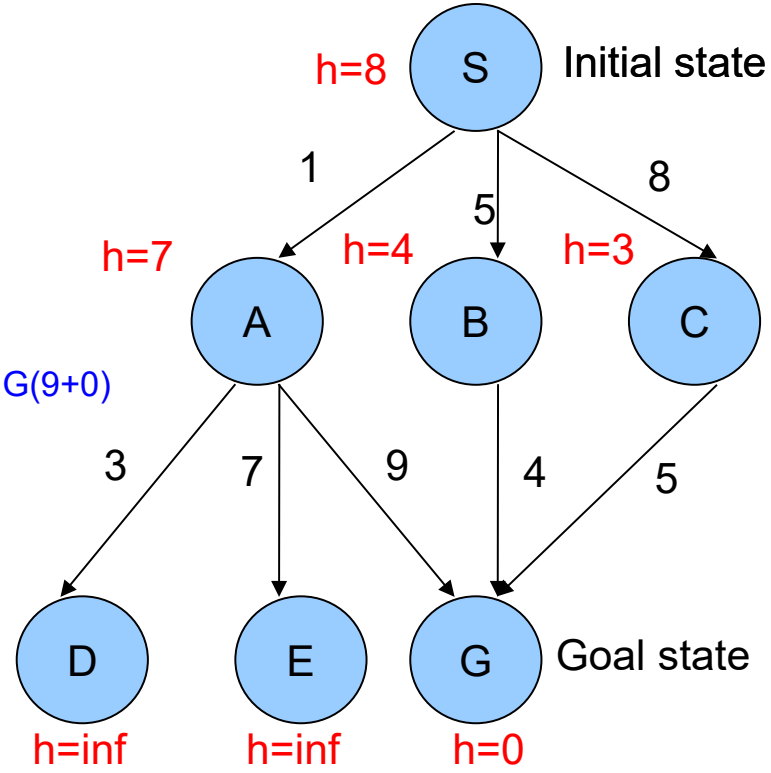


Recap and Examples

Example for A*:

OPEN	CLOSED
S(0+8)	-
A(1+7) B(5+4) C(8+3)	S(0+8)
B(5+4) C(8+3) D(4+inf) E(8+inf) G(10+0)	S(0+8) A(1+7)
C(8+3) D(4+inf) E(8+inf) G(9+0)	S(0+8) A(1+7) B(5+4)
C(8+3) D(4+inf) E(8+inf)	S(0+8) A(1+7) B(5+4) G(9+0)

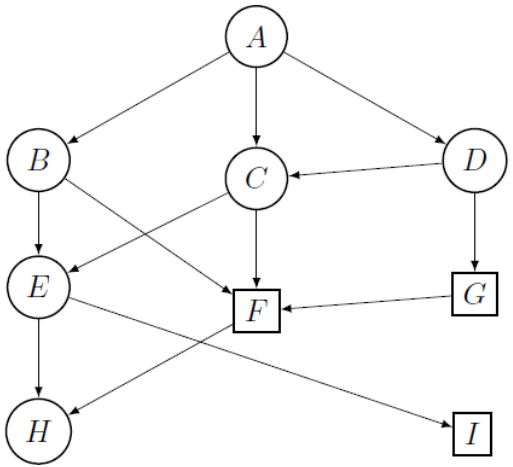
$G \rightarrow B \rightarrow S$



Break & Quiz

Q0: In the graph below, goal states are in rectangles. Which goal state will be found and returned by DFS and BFS? Break ties by alphabetical order (ie. letters that come earlier in the alphabet get higher priority).

A. BFS: F, DFS: F
B. BFS: F, DFS: I
C. BFS: I, DFS: F
D. BFS: F, DFS: G
E. BFS: G, DFS: I



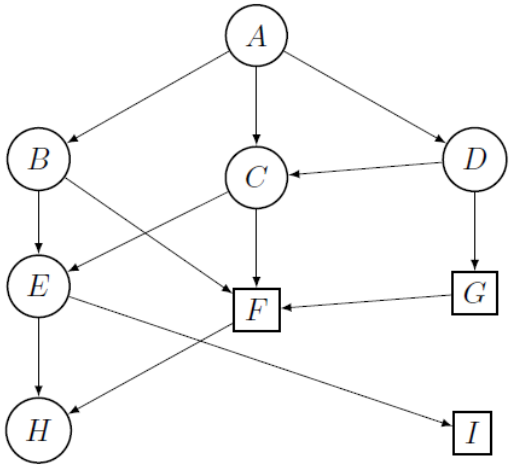
Break & Quiz

Q0: In the graph below, goal states are in rectangles. Which goal state will be found and returned by DFS and BFS? Break ties by alphabetical order (ie. letters that come earlier in the alphabet get higher priority).

A. BFS: F, DFS: F
B. BFS: F, DFS: I
C. BFS: I, DFS: F
D. BFS: F, DFS: G
E. BFS: G, DFS: I

In BFS, we process the nodes in order A,B,C,D,E,F which is a goal state.

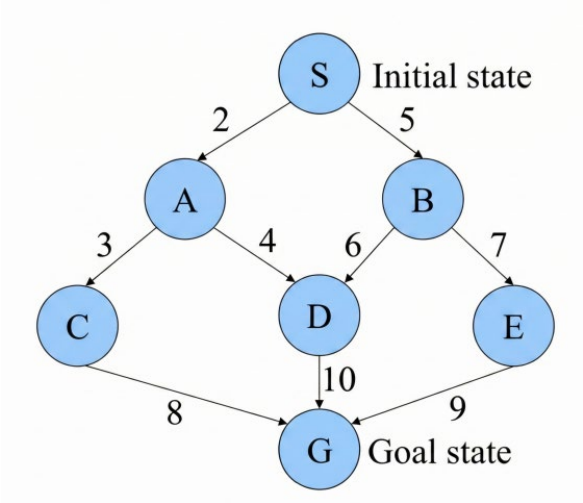
In DFS, we process the nodes in order A,B,E,H,I which is a goal state.



Break & Quiz

Q1: We are performing Uniform Cost Search (UCS) on the graph shown in the following figure. The initial state is S and the goal state G. What is the optimal cost to reach G from S using UCS?

A. 9
B. 10
C. 13
D. 16
E. 21

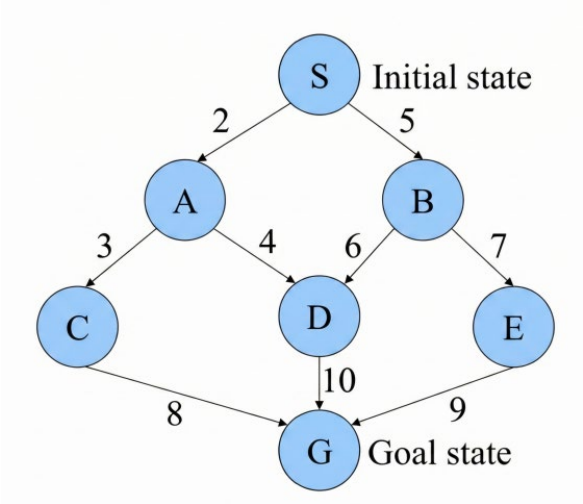


Break & Quiz

Q1: We are performing Uniform Cost Search (UCS) on the graph shown in the following figure. The initial state is S and the goal state G. What is the optimal cost to reach G from S using UCS?

A. 9
B. 10
C. 13
D. 16
E. 21

The optimal path is S-A-C-G and the cost for the path is 13.



Break & Quiz

Q 2: Let h_0 be an admissible heuristic for a search problem. Which of the following h will always be admissible?

A. $h(s) = \sqrt{h_0}(s)$
B. $h(s) = (h_0(s))^2$
C. $h(s) = h_0(s) - 1$
D. $h(s) = h_0(s) + 1$
E. None of the above

Break & Quiz

Q 2: Let h_0 be an admissible heuristic for a search problem. Which of the following h will always be admissible?

A. $h(s) = \sqrt{h_0}(s)$ (not admissible if h_0 <1)
B. $h(s) = (h_0(s))^2$ (not admissible if h_0 >1)
C. $h(s) = h_0(s) - 1$ (not admissible if h_0 <1)
D. $h(s) = h_0(s) + 1$ (not admissible because it's always $h_0(s) < h(s)$)
E. None of the above

Break & Quiz

Q3: Consider A∗ search on the grid below, where the initial state is A and the goal state is G. Valid moves are left, right, up, or down one square at a time, without wrapping around edges (so A → G is not a valid path, but A → D → G is a valid path). The cost g is the number of moves taken, and the heuristic h is the Manhattan distance to G:

A	B	C
D	E	F
G	H	I

What states remain in OPEN when the algorithm goal-checks G and terminates?

A. A,B,D,E.
B. A,B,D.
C. B,E.
D. D.
E. None of the above

Break & Quiz

Q3: Consider A∗ search on the grid below, where the initial state is A and the goal state is G. Valid moves are left, right, up, or down one square at a time, without wrapping around edges (so A → G is not a valid path, but A → D → G is a valid path). The cost g is the number of moves taken, and the heuristic h is the Manhattan distance to G:

A	B	C
D	E	F
G	H	I

What states remain in OPEN when the algorithm goal-checks G and terminates?

A. A,B,D,E.

B. A,B,D.

c. B,E.

D. D.

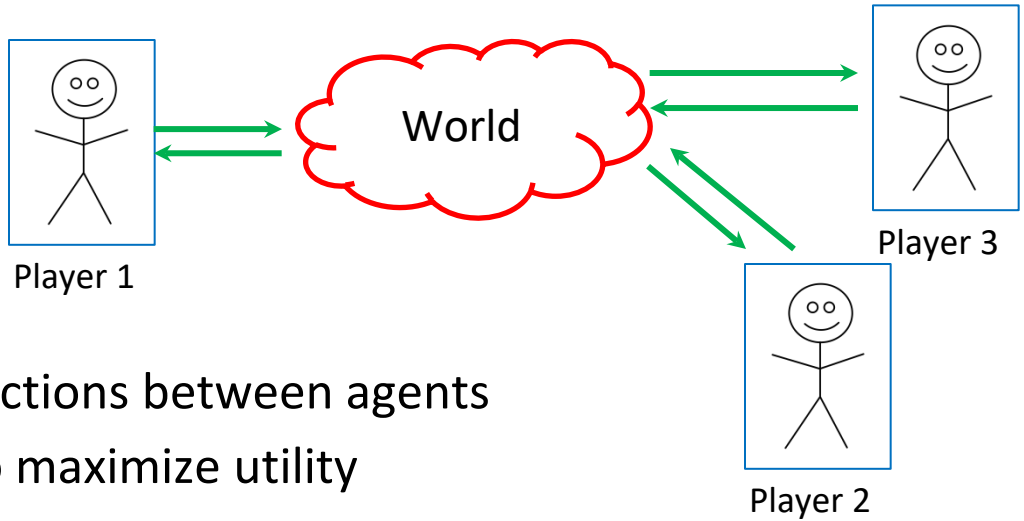
E. None of the above

The algorithm starts by expanding A and adding B,D to OPEN. Since g(D) + h(D) = 1 + 1 = 2 and g(B) + h(B) = 1 + 3 = 4, the algorithm expands D next, and now OPEN contains B,E,G. Since g(E) + h(E) = 2 + 2 = 4 and g(G) + h(G) = 2 + 0 = 2, the algorithm expands G next, and terminates with open containing B,E.

Games

Games Setup

Games setup: **multiple** agents



- Now: interactions between agents
- Still want to maximize utility
- **Strategic** decision making.

Normal Form Game

Mathematical description of simultaneous games.

- n players {1,2,…,n}
- Player i chooses strategy a_i from action space A_i.
- Strategy profile: $a = (a_1, a_2, \ldots, a_n)$
- Player i gets rewards $u_i(a)$
  - **Note**: reward depends on other players!

- We consider the simple case where all reward functions are common knowledge.

Example of Normal Form Game

Ex: Prisoner's Dilemma

Player 2 Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

- 2 players, 2 actions: yields 2x2 payoff matrix
- Strategy set: {Stay silent, betray}

Strictly Dominant Strategies

Let's analyze such games. Some strategies are better than others!

- Strictly dominant strategy: if a_i strictly better than b *regardless* of what other players do, a_i is **strictly dominant**
- I.e., $u_i(a_i, a_{-i}) > u_i(b, a_{-i}), \forall b \neq a_i, \forall a_{-i}$

All of the other entries of *a* excluding *i*

- Sometimes a dominant strategy does not exist!

Dominant Strategy Equilibrium

a^* is a (strictly) dominant strategy equilibrium (DSE), if every player *i* has a strictly dominant strategy a_i^*

- Rational players will play at DSE, if one exists.

Player 2 / Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

Dominant Strategy: Absolute Best Responses

Player *i*'s best response to strategy to $a_{-i} BR(a_{-i}) = \arg\max_b u_i(b, a_{-i})$

BR(player2=silent) = betray

BR(player2=betray) = betray

Player 1 \ Player 2	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

a_i^* is the dominant strategy for player *i*, if

$$a_i^* = BR(a_{-i}), \forall\ a_{-i}$$

Dominant Strategy Equilibrium

Dominant Strategy Equilibrium does not always exist.

Player 2 Player 1	*L*	*R*
L	2, 1	0, 0
R	0, 0	1, 2

Nash Equilibrium

*a** is a Nash equilibrium if no player has an incentive to **unilaterally deviate**

$$u_i(a_i^*, a_{-i}^*) \geq u_i(a_i, a_{-i}^*) \quad \forall a_i \in A_i$$

Player 2 / Player 1	*L*	*R*
L	2, 1	0, 0
R	0, 0	1, 2

Nash Equilibrium: Best Response to Each Other

- Compared to DSE: **a DSE is a NE**, the other direction is generally not true:

$$a_i^* = BR(a_{-i}), \forall\, a_{-i}$$

- Pure Nash equilibrium:
  - A **pure strategy** is a deterministic choice (no randomness).
  - Later: we will consider **mixed** strategies
  - In pure Nash equilibrium, players can only play pure strategies.

Break & Quiz

Q4: Consider the following simultaneous-move game between players 1 and 2 with payoff matrix:

Player1\Player2	*L*	*R*
U	(4, 1)	(2, 2)
D	(3, 3)	(1, 4)

Which strategy is strictly dominant for player 2?

A. *L.*

B. *R*.

C. Both *L* and *R*.

D. Neither *L* nor *R*.

E. It depends on player 1's action

Break & Quiz

Q4: Consider the following simultaneous-move game between players 1 and 2 with payoff matrix:

Player1\Player2	*L*	*R*
U	(4, 1)	(2, 2)
D	(3, 3)	(1, 4)

Which strategy is strictly dominant for player 2?

A. *L.*

B. *R.*

C. Both *L* and *R*.

D. Neither *L* nor *R*.

E. It depends on player 1's action

Compare player 2's payoffs:

- If player 1 plays U: $u_2(L) = 1$, $u_2(R) = 2$.
- If player 1 plays D: $u_2(L) = 3$, $u_2(R) = 4$.

In both cases $u_2(R) > u_2(L)$, so R strictly dominates L.

Hence the strictly dominant strategy for player 2 is R.

Break & Quiz

Q5: Consider the following payoff matrix for a two-player normal form game:

P1\P2	*C*	*D*
A	(0, 1)	(1, 0)
B	(1, 0)	(0, 2)

Which of the following is a pure Nash equilibrium?

A. (A,C)
B. (A,D)
C. (B,C)
D. (B,D)
E. The game does not have a pure Nash equilibrium.

Break & Quiz

Q5: Consider the following payoff matrix for a two-player normal form game:

P1\P2	*C*	*D*
A	(0, 1)	(1, 0)
B	(1, 0)	(0, 2)

Which of the following is a pure Nash equilibrium?

A. (A,C)

B. (A,D)

C. (B,C)

D. (B,D)

E. The game does not have a pure Nash equilibrium.

Pure Nash Equilibrium may not exist

So far, pure strategy: each player picks a deterministic strategy. But:

Player 2 Player 1	*rock*	paper	*scissors*
rock	0, 0	-1, 1	1, -1
paper	1, -1	0, 0	-1, 1
scissors	-1, 1	1, -1	0, 0

Mixed Strategies

Can also randomize actions: "**mixed**"

- Player *i* assigns probabilities x_i to each action

$$x_i(a_i), \text{ where } \sum_{a_i \in A_i} x_i(a_i) = 1, x_i(a_i) \geq 0$$

- Now consider **expected rewards**

$$u_i(x_i, x_{-i}) = E_{a_i \sim x_i, a_{-i} \sim x_{-i}} u_i(a_i, a_{-i}) = \sum_{a_i} \sum_{a_{-i}} x_i(a_i) x_{-i}(a_{-i}) u_i(a_i, a_{-i})$$

Mixed Strategy Nash Equilibrium

Consider the mixed strategy $x^* = (x_1^*, \ldots, x_n^*)$

- This is a **Nash equilibrium** if

$$u_i(x_i^*, x_{-1}^*) \geq u_i(x_i, x_{-i}^*) \quad \forall x_i \in \Delta_{A_i}, \forall i \in \{1, \ldots, n\}$$

Better than doing anything else, "**best response**"

Space of probability distributions over strategies.

- Intuition: nobody can **increase expected reward** by changing only their own strategy.

Mixed Strategy Nash Equilibrium

Example: $x_1^*(\cdot) = x_2^*(\cdot) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$

Player 2 Player 1	*rock*	paper	*scissors*
rock	0, 0	-1, 1	1, -1
paper	1, -1	0, 0	-1, 1
scissors	-1, 1	1, -1	0, 0

Finding Mixed NE in 2-Player 2-action Zero-Sum Game

Example: Two Finger Morra. Show 1 or 2 fingers. The “even player” wins if the sum is even, and vice versa.

odd / even	*f1*	f2
f1	2, -2	-3, 3
f2	-3, 3	4, -4

Finding Mixed NE in 2-Player 2-action Zero-Sum Game

Two Finger Morra. Two-player zero-sum game. No pure NE:

odd / even	*f1*	f2
f1	2, -2	-3, 3
f2	-3, 3	4, -4

Finding Mixed NE in 2-Player 2-action Zero-Sum Game

Suppose odd's mixed strategy at NE is (q, 1-q), and even's (p, 1-p)

By definition, p is best response to q: $u_1(p,q) \geq u_1(p',q) \forall p'$.

Note $u_1(p,q) = pu_1(f_1,q) + (1-p)u_1(f_2,q)$

- Players only mix strategies if the expected payoffs are equal.
- If one strategy was better, they would never mix — they'd just pick the best!
  → $u_1(f_1,q) = u_1(f_2,q)$

- Average is no greater than components
  → $u_1(p,q) = u_1(f_1,q) = u_1(f_2,q)$

		q	1-q
	even \ odd	*f1*	f2
p	*f1*	2, -2	-3, 3
1-p	f2	-3, 3	4, -4

Finding Mixed NE in 2-Player 2-action Zero-Sum Game

$$u_1(f_1, q) = u_1(f_2, q)$$
$$2q + (-3)(1-q) = (-3)q + 4(1-q)$$
$$q = \frac{7}{12}$$

Similarly, $u_2(p, f_1) = u_2(p, f_2)$

$$p = \frac{7}{12}$$

At this NE, even gets -1/12, odd gets 1/12.

		q	1-q
	odd / even	*f1*	f2
p	*f1*	2, -2	-3, 3
1-p	f2	-3, 3	4, -4

Properties of Nash Equilibrium

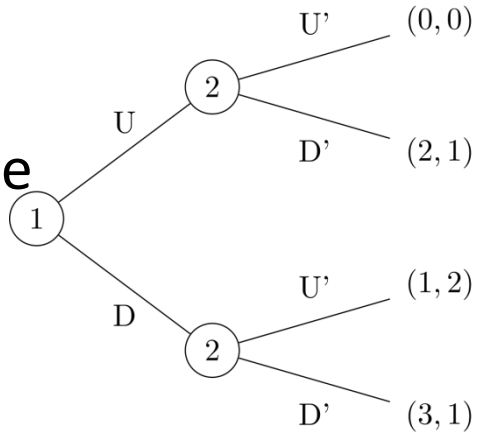
Major result: (John Nash ’51)

- Every finite (players, actions) game has at least one Nash equilibrium
  - But not necessarily **pure** (i.e., deterministic strategy)
- Could be more than one
- Searching for Nash equilibria: computationally **hard**.
  - Exception: two-player zero-sum games (can be found with linear programming).

Sequential-Move Games

More complex games with multiple moves

- Instead of normal form, **extensive form**
- Represent with a **tree**
- **Rewards at leaves**
- Find strategies: perform search over the tree



Wiki

- Nash equilibrium still well-defined
  - Backward induction

Our Approach

We find the minimax value/strategy bottom up

- Minimax value: score of terminal node when both players play optimally
  - **Max's** turn, take max of children
  - **Min's** turn, take min of children

- Can implement this as depth-first search: **minimax algorithm**

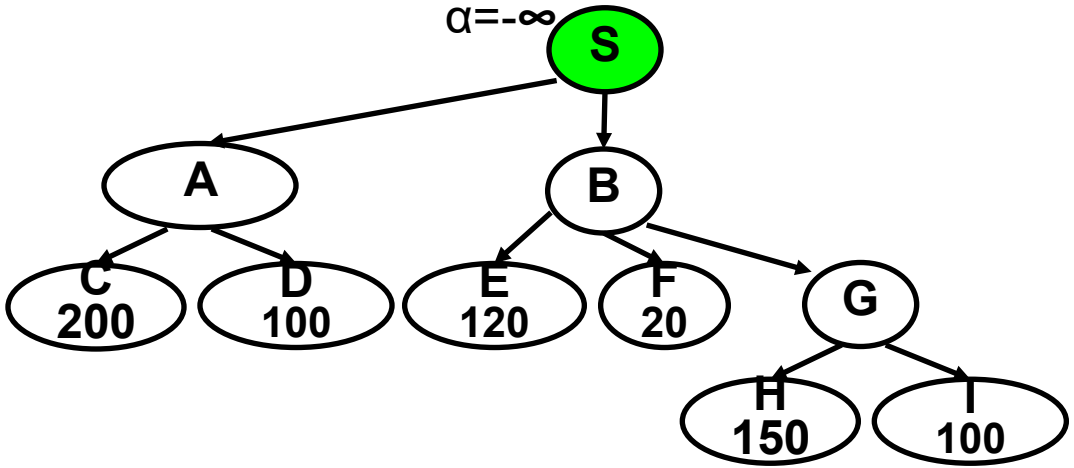
Minimax algorithm in execution

max

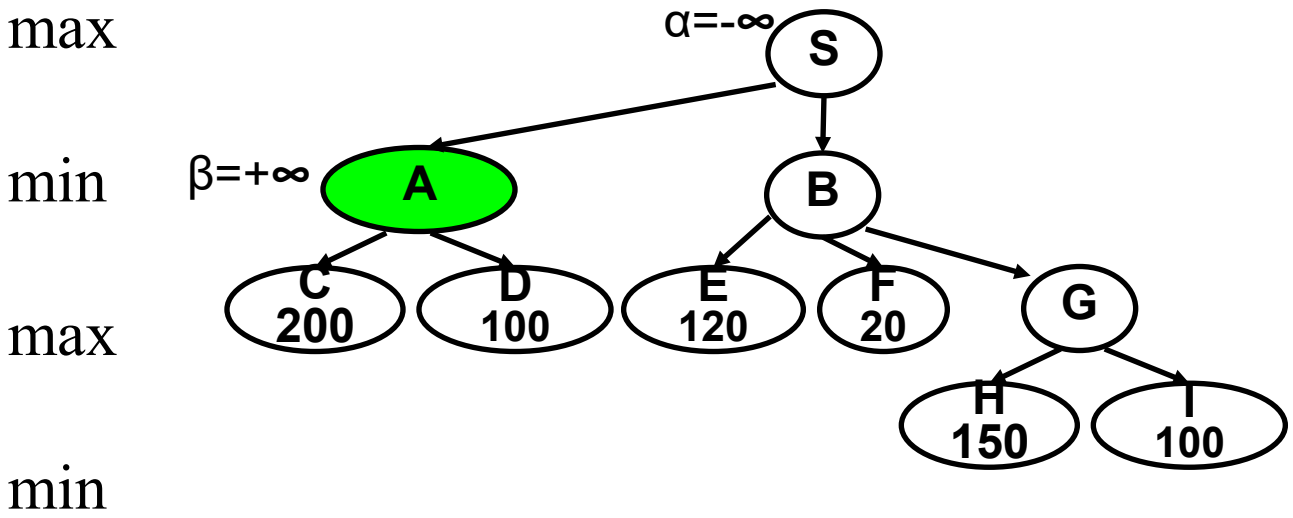
min

max

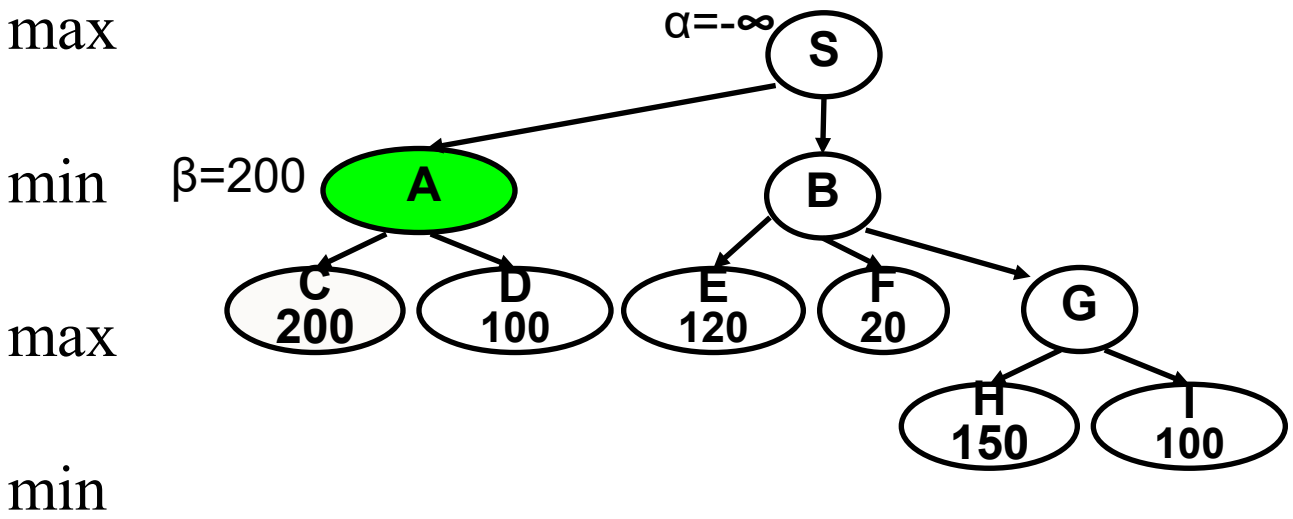
min



Minimax algorithm in execution

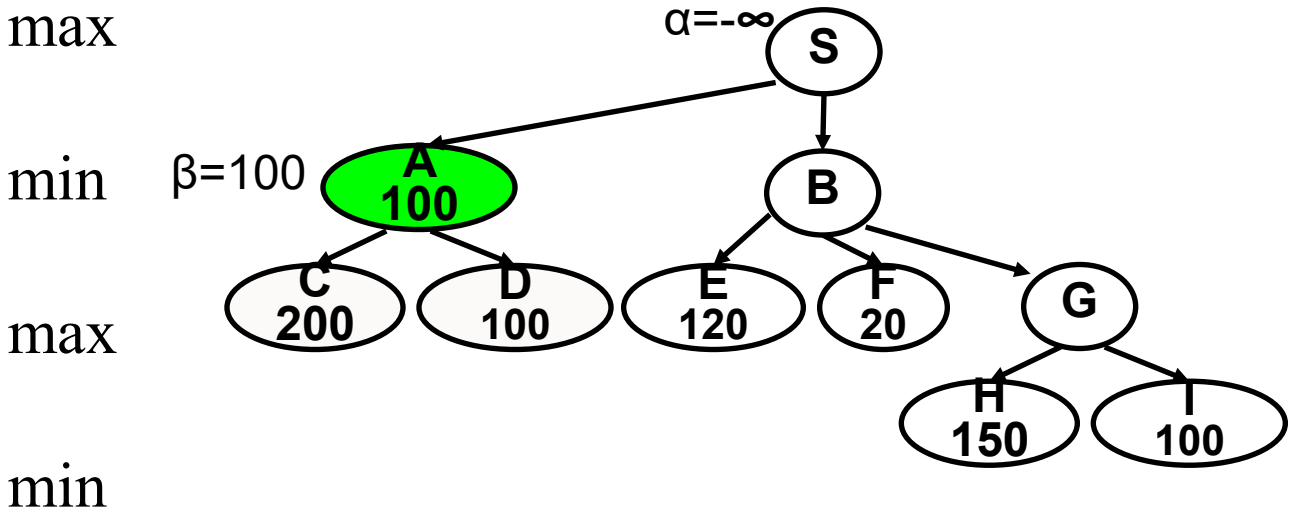


Minimax algorithm in execution

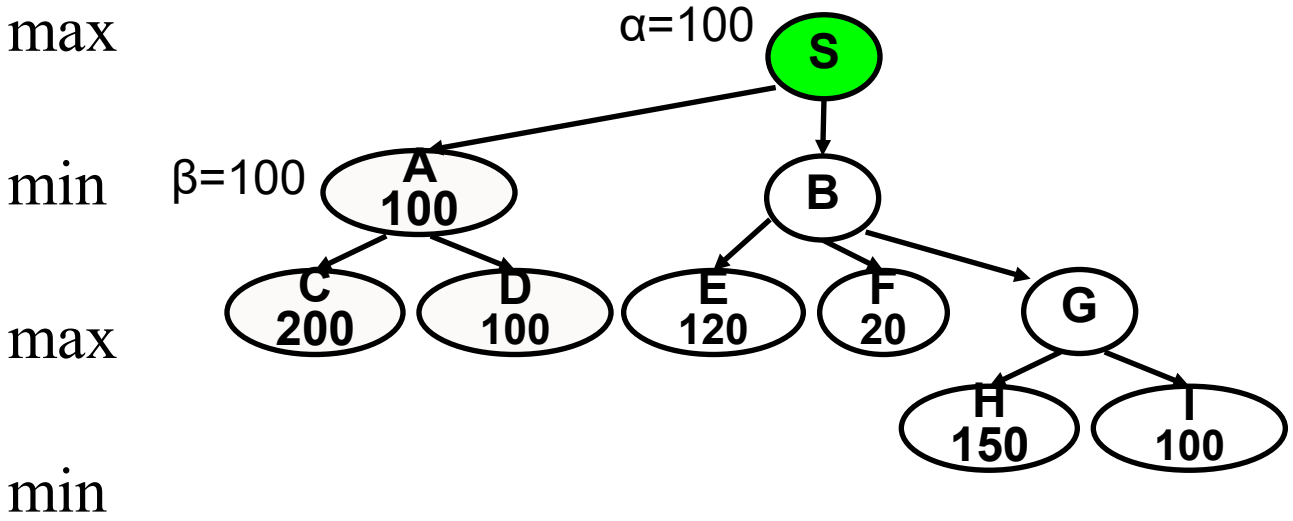


The execution on the terminal nodes is omitted.

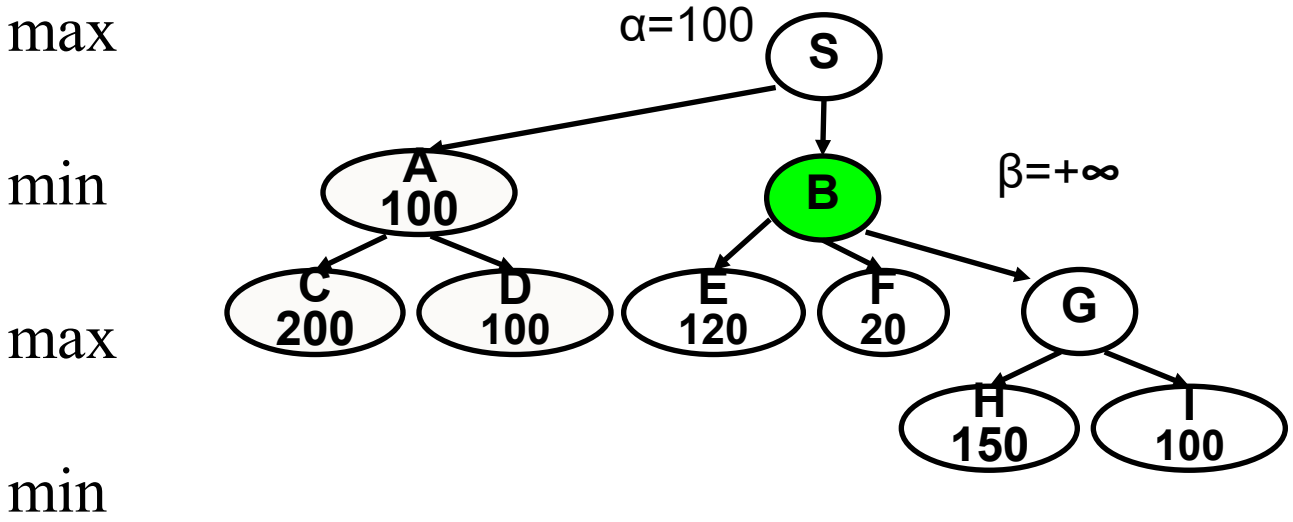
Minimax algorithm in execution



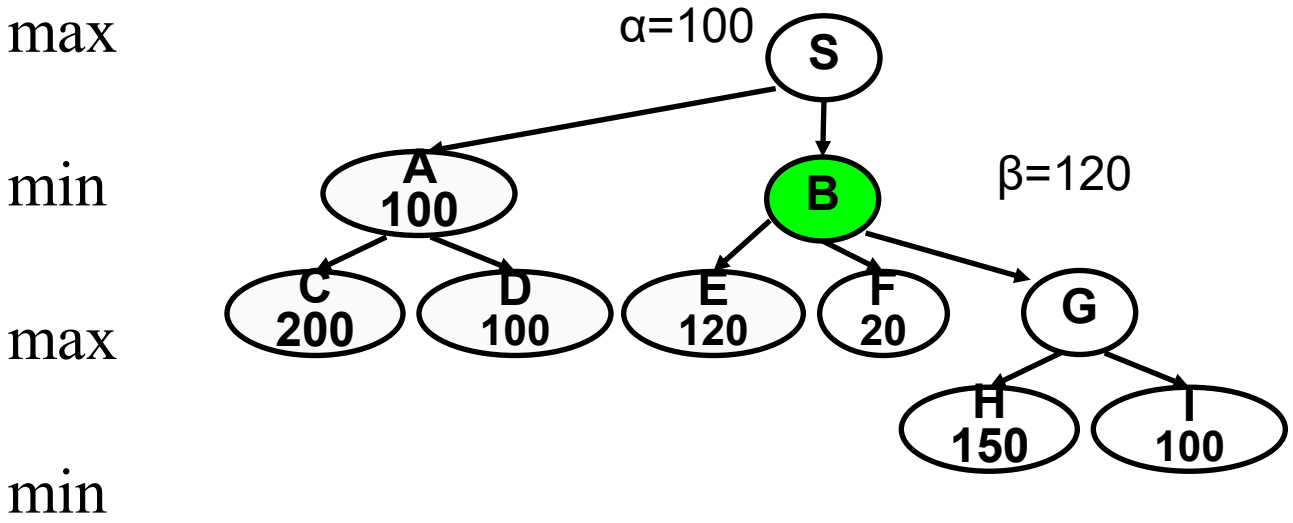
Minimax algorithm in execution



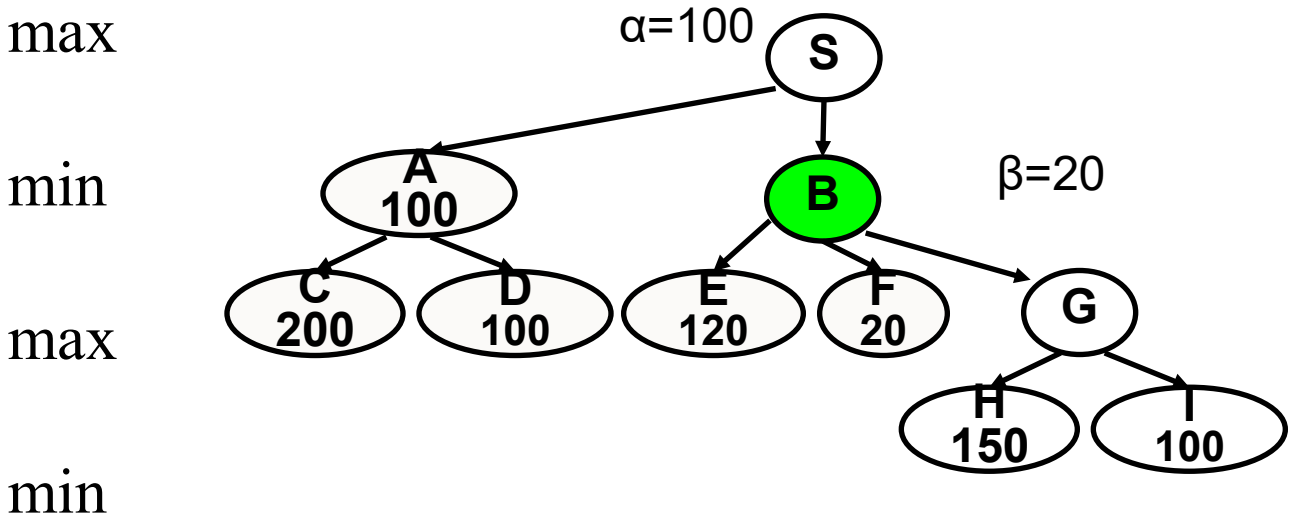
Minimax algorithm in execution



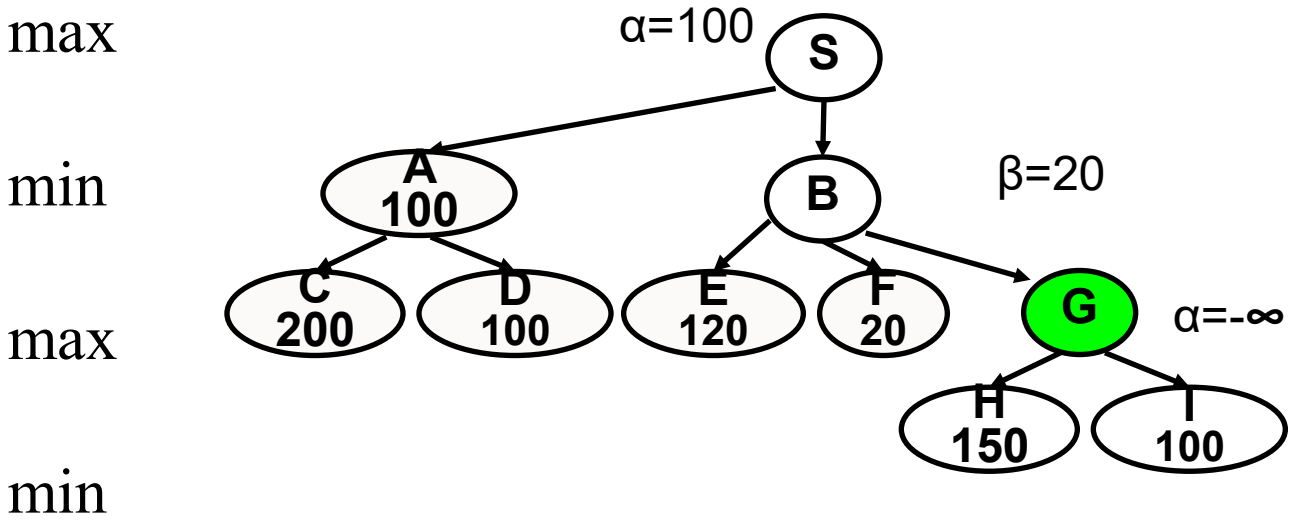
Minimax algorithm in execution



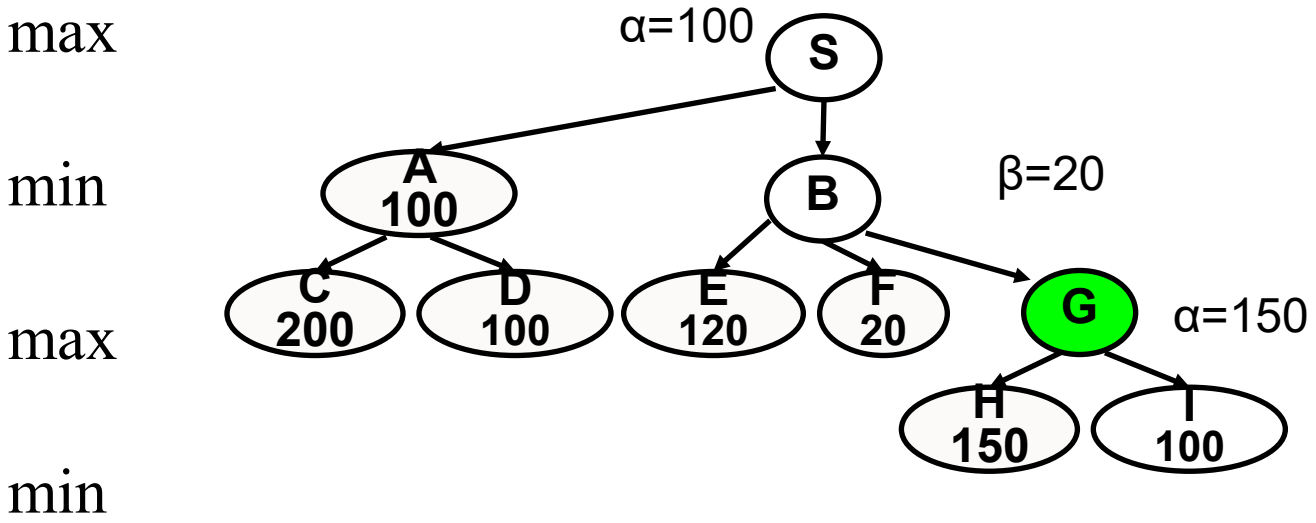
Minimax algorithm in execution



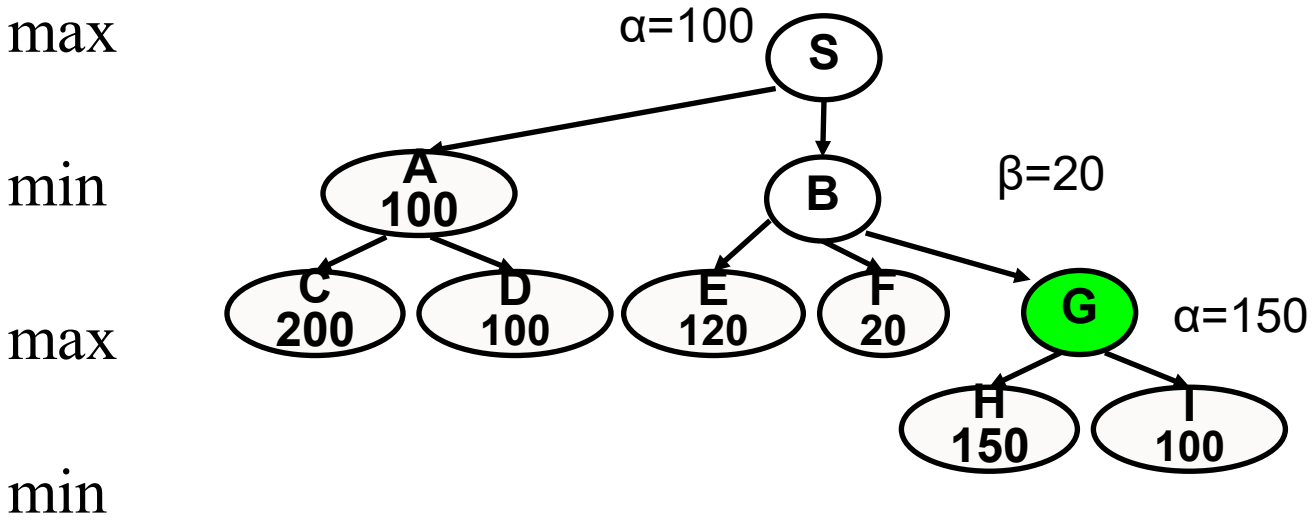
Minimax algorithm in execution



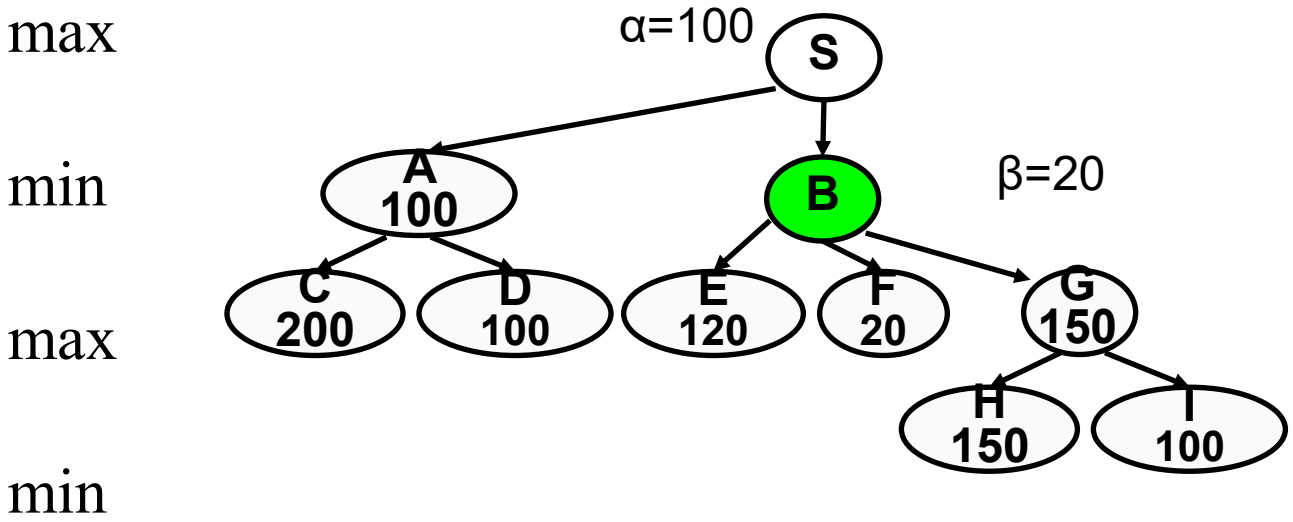
Minimax algorithm in execution



Minimax algorithm in execution



Minimax algorithm in execution



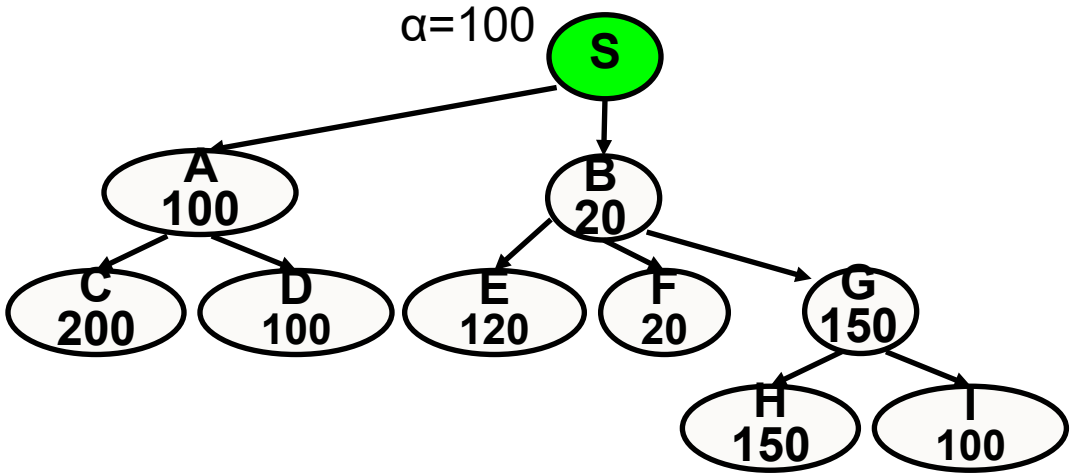
Minimax algorithm in execution

max

min

max

min



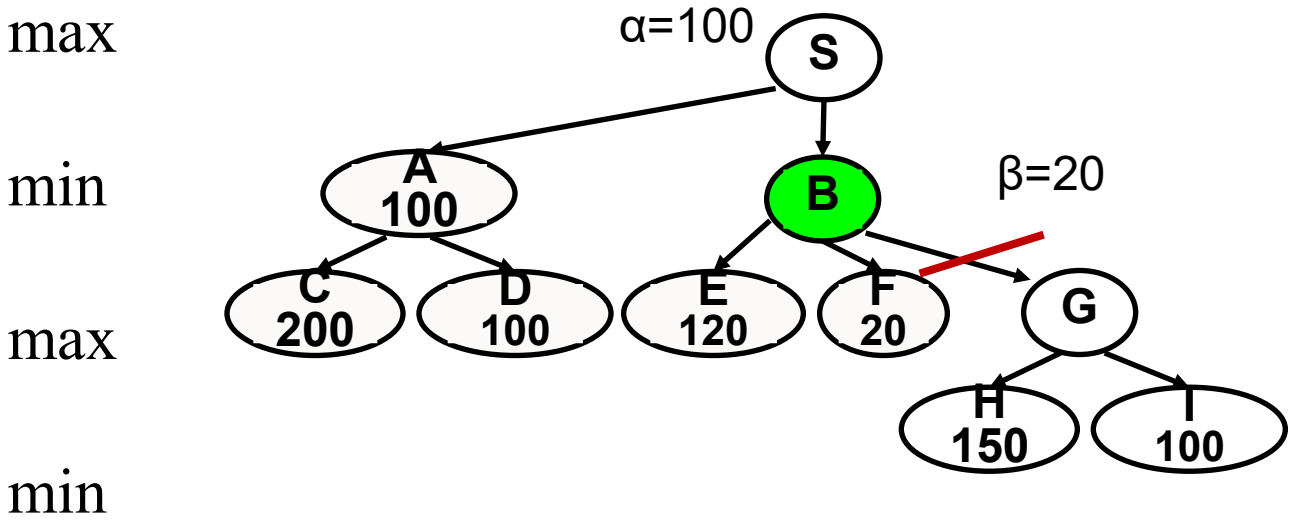
Can We Do Better?

One **downside**: we had to examine the entire tree

An idea to speed things up: **pruning**

- Goal: want the same minimax value, but faster
- We can get rid of bad branches

Minimax algorithm in execution



Break & Quiz

Q 6: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- B. 65
- C. 41
- D. 64
- E. 2

Break & Quiz

Q 6: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- **B. 65**
- C. 41
- D. 64
- E. 2

Break & Quiz

Q 6: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

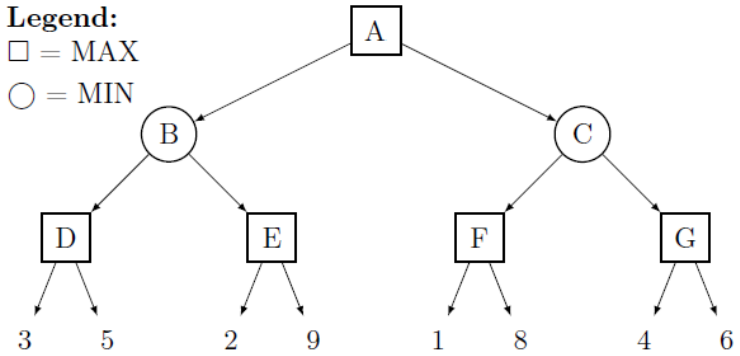
How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- **B. 65** (1 + 4 + 4*3 + 4*3*2 + 4*3*2 = 65. Note the root and leaf nodes.)
- C. 41
- D. 64
- E. 2

Break & Quiz

Q 7: Consider the game tree shown in the figure below. The root node (A) represents the current state of the game where it is the MAX player's turn.
Assuming both players play optimally according to the Minimax algorithm, what is the value of the root node A?

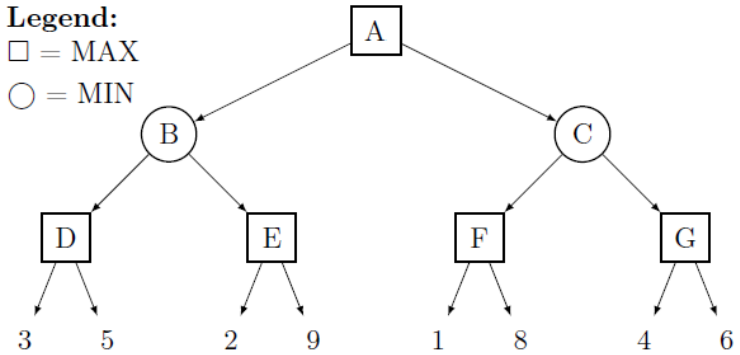
A. 4
B. 5
C. 6
D. 8
E. 9



Break & Quiz

Q 7: Consider the game tree shown in the figure below. The root node (A) represents the current state of the game where it is the MAX player's turn.
Assuming both players play optimally according to the Minimax algorithm, what is the value of the root node A?

A. 4
B. 5
C. 6
D. 8
E. 9

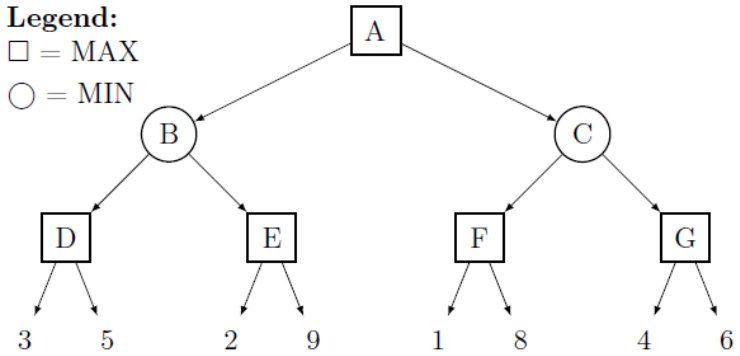


Break & Quiz

Q 7: Consider the game tree shown in the figure below. The root node (A) represents the current state of the game where it is the MAX player's turn.
Assuming both players play optimally according to the Minimax algorithm, what is the value of the root node A?

A. 4
B. 5
C. 6
D. 8
E. 9

The Minimax algorithm propagates values from the leaf nodes up to the root.
1. Level 2 (MAX Nodes): The MAX player chooses the highest value among the children.
• Node D: max(3, 5) = 5
• Node E: max(2, 9) = 9
• Node F: max(1, 8) = 8
• Node G: max(4, 6) = 6
2. Level 1 (MIN Nodes): The MIN player chooses the lowest value among the children passed up from Level 2.
• Node B: min(D,E) = min(5, 9) = 5
• Node C: min(F,G) = min(8, 6) = 6
3. Level 0 (Root Node): The MAX player chooses the highest value among the children passed up from Level 1.
• Node A: max(B,C) = max(5, 6) = 6



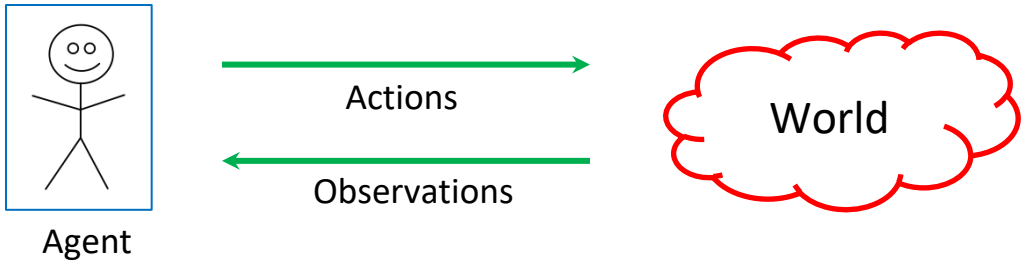
Reinforcement Learning

Building The Theoretical Model

Basic setup:

Set of states, S

Set of actions A



Information: at time t, observe state $s_t \in S$. Get reward r_t

Agent makes choice $a_t \in A$. State changes to $s_{t+1,}$ continue

Goal: find a map from **states to actions** maximize rewards.

A “policy”

Markov Decision Process (MDP)

The formal mathematical model:

State set S. Initial state s_0. **Action set** A

State transition model: $P(s_{t+1}|s_t, a_t)$

- Markov assumption: transition probability only depends on s_t and a_t, and not previous actions or states.

Reward function: *r*(s_t)

Policy: $\pi(s) : S \rightarrow, A$ action to take at a particular state.

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots$$

Defining the Optimal Policy

For policy π, **expected utility** over all possible state sequences from s_0 produced by following that policy:

Utility of sequence

$$V^{\pi}(s_0) = \sum_{\text{sequences starting from } s_0} P(\text{sequence})U(\text{sequence})$$

Probability of sequence when following π

Called the **value function** (for π, s_0)

Discounting Rewards

One issue: these are infinite series. **Convergence**?

Solution

$$U(s_0, s_1 \ldots) = r(s_0) + \gamma r(s_1) + \gamma^2 r(s_2) + \ldots = \sum_{t \geq 0} \gamma^t r(s_t)$$

Discount factor γ between 0 and 1

- Set according to how important **present** is VS **future**
- Note: has to be less than 1 for convergence

Obtaining the Optimal Policy

Assume, we know the expected utility of an action.

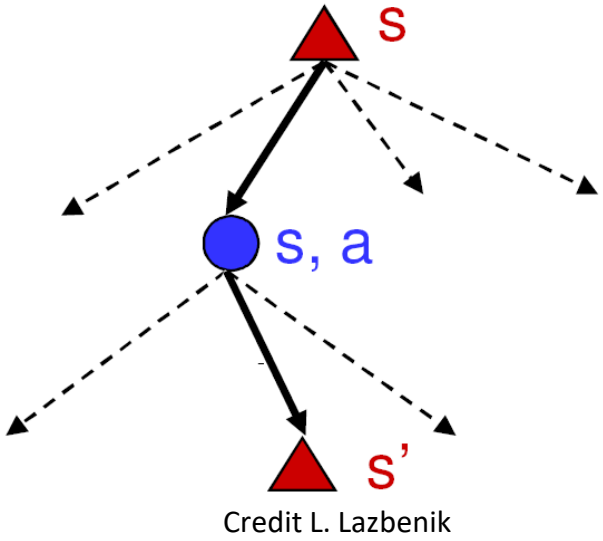
So, to get the optimal policy, compute

$$\pi^*(s) = \text{argmax}_a \sum_{s'} P(s'|s,a)V^*(s')$$

All the states we could go to

Transition probability

Expected rewards

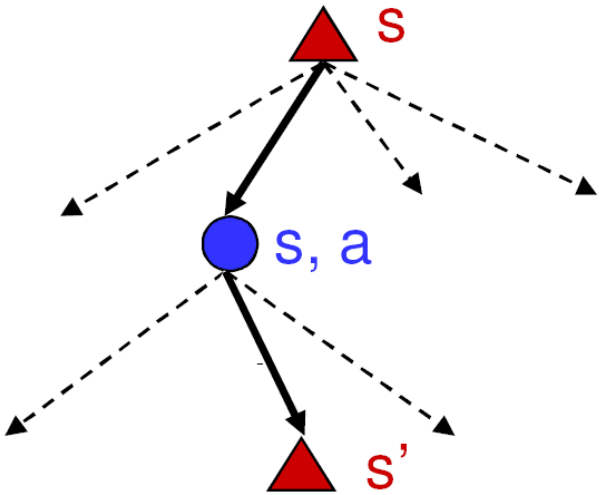


Credit L. Lazbenik

Bellman Equations

Let's walk over one step for the value function:

$$V^*(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a)V^*(s')$$



Credit L. Lazbenik

Current state reward

Discounted expected future **rewards**

Richard Bellman: Inventor of dynamic programming.

Value Iteration

Q: how do we find $V^*(s)$?

Why do we want it? Can use it to get the best policy

Know: reward $r(s)$, transition probability $P(s'|s,a)$

- Knowing r and P is the "planning" problem. In reality r and P must be estimated from interactions : "reinforcement learning"

Also know $V^*(s)$ satisfies Bellman equation (recursion above)

A: Use the property. Start with $V_0(s)=0$. Then, update

$$V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$$

Q-Learning

- Our **next** reinforcement learning algorithm.
- Does not require knowing *r* or *P*. Learn from data of the form:$\{(s_t, a_t, r_t, s_{t+1})\}$.
- Learns an action-value function *Q**(*s*,*a*) that tells us the expected value of taking *a* in state *s*.
  - Note: $V^*(s) = \underset{a}{max} Q^*(s, a)$.
- Optimal policy is formed as $\pi^*(s) = \arg\underset{a}{max} Q^*(s, a)$

Q-Learning

Estimate Q*(*s*,*a*) from data $\{(s_t, a_t, r_t, s_{t+1})\}$:

1. Initialize Q(.,.) arbitrarily (eg all zeros)
    1. Except terminal states $Q(s_{terminal},.)=0$
2. Iterate over data until Q(.,.) converges:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$

Learning rate

Example

A -10	G 10
S	B

discount factor $\gamma = 0.9$

learning rate $a = 0.1$

Example

A -10	G 10
S	B

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$

discount factor $\gamma = 0.9$

learning rate $a = 0.1$

	left	right	up	down
S			-1	
A				
B				
G				

$$Q(S, up) = (1-0.1) * 0 + 0.1(-10) = -1$$

Example

A -10	G 10
S	B

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$

discount factor $\gamma = 0.9$

learning rate $a = 0.1$

	left	right	up	down
S		0	-1	
A				
B				
G				

$$Q(S, right) = (1-0.1) * 0 + 0.1(0 + 0.9 * 0) = \ 0$$

Example

A^{-10}	G^{10}
S	B

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$

discount factor $\gamma = 0.9$

learning rate $a = 0.1$

	left	right	up	down
S		0	-1	
A				
B			1	
G				

$$Q(S, right) = (1-0.1) * 0 + 0.1(0 + 0.9 * 0) = \ 0$$

$$Q(B, up) = (1-0.1) * 0 + 0.1(10) = \ 1$$

Example

A -10	G 10
S	B

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$

discount factor $\gamma = 0.9$

learning rate $a = 0.1$

	left	right	up	down
S		0.09	-1	
A				
B			1.9	
G				

$$Q(S, right) = (1-0.1) * 0 + 0.1(0 + 0.9 * 1) = \ 0.09$$

$$Q(B, up) = (1-0.1) * 1 + 0.1(10) = \ 1.9$$

Exploration Vs. Exploitation

General question!

Exploration: take an action with unknown consequences

- **Pros**:
  - Get a more accurate model of the environment
  - Discover higher-reward states than the ones found so far
- **Cons**:
  - When exploring, not maximizing your utility
  - Something bad might happen

Exploitation: go with the best strategy found so far

- **Pros**:
  - Maximize reward as reflected in the current utility estimates
  - Avoid bad stuff
- **Cons**:
  - Might prevent you from discovering the true optimal strategy

Q-Learning: ε-Greedy Behavior Policy

Getting data with both **exploration and exploitation**

With probability ε, take a random action; else the action with the highest (current) Q(*s*,*a*) value.

$$a = \begin{cases} \text{argmax}_{a \in A} Q(s, a) & \text{uniform}(0, 1) > \epsilon \\ \text{random } a \in A & \text{otherwise} \end{cases}$$

Q-learning Algorithm

Input: step size α, exploration probability ϵ

1. set $Q(s,a) = 0$ for all s, a.
2. For each episode:
3. Get initial state s.
4. While (s not a terminal state):
5. Perform a = ϵ-greedy(Q, s), receive r, s'
6. $$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$
7. $s \leftarrow s'$
8. End While
9. End For

Explore: take action to see what happens.

Update action-value based on result.

Deep Q-Learning

Q-tables work great for small, discrete state spaces.

How do we get Q(*s*,*a*) with a large number of states?

Deep Q-learning uses a **neural network** to approximate Q(s,a)

- Let $Q_\theta : S \times A \to \mathbb{R}$ be the neural network with weights and biases denoted θ.

Training is similar to supervised regression:

- (s, a) as input and compute target $y = r(s) + \gamma \max_{a'} Q_\theta(s', a')$.
- Note that output of the neural network is used in the target/label.
- Loss function: $\mathcal{L}(\theta) = (y - Q_\theta(s, a))^2$

Problem with Deep Q-Learning

This doesn't work well! Training is **very unstable**

- Correlated samples

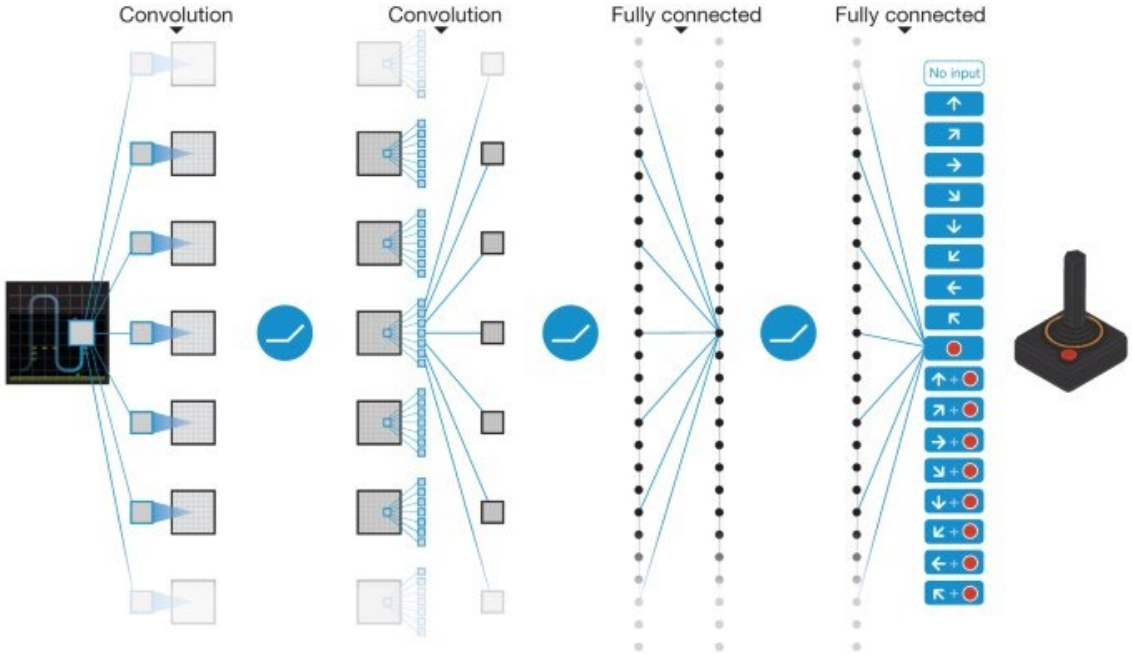
  — consecutive experiences are highly correlated

- Moving target

  — the target y changes every time we update θ

Deep Q-Networks (DQN)

Deep Q-Networks stabilized Deep Q-Learning.



Mnih et al, "Human-level control through deep reinforcement learning"

DQN Key Innovations

1. **Experience Replay**:
   - Store all transitions in a replay buffer D
   - At each training step, sample a random mini-batch from D
   - Use this mini-batch to compute the gradient update
2. Separate **target network (θ^-)** to compute targets:
   - **Online Network (θ)**
     - Updated every step via SGD
     - Used to select actions (ε-greedy)
   - **Target Network (θ^-)**
     - Frozen copy of θ
     - Updated: $\theta^- \leftarrow \theta$ every C steps

Break & Quiz

Q8: Supposed you have the following environment where a robot can move. The robot can only move horizontally or vertically by taking the following actions u (up), d (down), l (left), r (right) and you have the following information:

s_0	s_1	s_2
s_3	s_4	s_5
s_6	s_7	s_8

1. The reward in s_3 is 2
2. The transition probabilities are: P(s_0 | s_3, r) = 0.1 and P(s_4 | s_3, r) = 0.7 and P(s_6 | s_3, r) = 0.2
3. Currently, V (s_0) = 0 and V (s_4) = 10 and V (s_6) = 5
4. The discount factor is 0.5

Remember the update for value iteration is $V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$

Assuming the maximizing action taken from s_3 is r, what is the value V (s_3) for state s_3, after a single iteration of value iteration?

A. 1
B. 2
C. 4
D. 5
E. 6

Break & Quiz

Q8: Supposed you have the following environment where a robot can move.
The robot can only move horizontally or vertically by taking the following actions u (up), d (down), l (left), r (right) and you have the following information:

s_0	s_1	s_2
s_3	s_4	s_5
s_6	s_7	s_8

1. The reward in s_3 is 2
2. The transition probabilities are: P(s_0 | s_3, r) = 0.1 and P(s_4 | s_3, r) = 0.7 and P(s_6 | s_3, r) = 0.2
3. Currently, V (s_0) = 0 and V (s_4) = 10 and V (s_6) = 5
4. The discount factor is 0.5

Remember the update for value iteration is $V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$

Assuming the maximizing action taken from s_3 is r, what is the value V (s_3) for state s_3, after a single iteration of value iteration?

A. 1
B. 2
C. 4
D. 5
E. 6

V (s_3) = 2 + 0.5 × (0.1 × 0 + 0.7 × 10 + 0.2 × 5) = 6

Break & Quiz

Q 9: Which of the following statements about reinforcement learning is False?

A. With a discount factor of $\gamma = 1$, the value function may fail to converge
B. The Markov property states that the next state depends only on the current state and action, not on the history of previous states.
C. In Q-learning, the agent must know the transition probabilities to update the Q-values.
D. The credit assignment problem refers to the difficulty of determining how much each action contributed to the final outcome.
E. In value iteration, the value function is updated by applying the Bellman equation repeatedly.

Break & Quiz

Q 9: Which of the following statements about reinforcement learning is False?

A. With a discount factor of $\gamma = 1$, the value function may fail to converge

B. The Markov property states that the next state depends only on the current state and action, not on the history of previous states.

C. In Q-learning, the agent must know the transition probabilities to update the Q-values.

D. The credit assignment problem refers to the difficulty of determining how much each action contributed to the final outcome.

E. In value iteration, the value function is updated by applying the Bellman equation repeatedly.

Advanced Search-Optimization

Hill Climbing Algorithm

Pseudocode:

1. Pick initial state *s*
2. Pick *t* in **neighbors**(*s*) with the best ***f***(*t*)
3. if ***f***(*t*) is not better than ***f***(*s*) THEN stop, return *s*
4. *s* ← *t*. goto 2.

What could happen? **Local optima!**

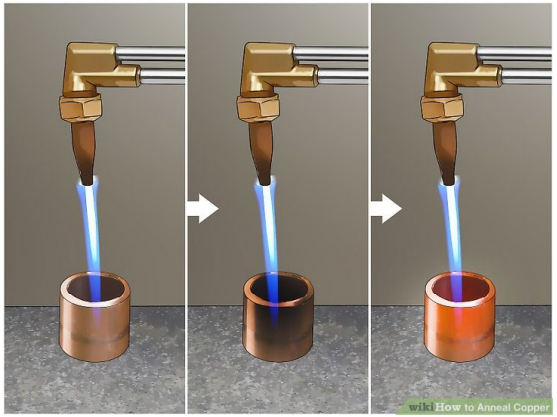
Simulated Annealing

A more sophisticated optimization approach.

Idea: allow some downhill moves at first, then be pickier over time

Pseudocode:

Pick initial state s; T=1
For k = 0 through K:
$T \leftarrow T*0.99$ (cool down)
Pick a random neighbour $t \leftarrow$ neighbor(s)
If $f(t)$ better than $f(s)$, then $s \leftarrow t$
Else with prob. $P(f(s), f(t), T)$ still do $s \leftarrow t$
Output: the best state ever seen



wikihow.com

Simulated Annealing: Picking Probability

How do we pick probability P? Note 3 parameters.

Decrease with time

Decrease with gap $|f(s) - f(t)|$:

$$\exp\left(-\frac{|f(s)-f(t)|}{Temp}\right)$$

Temperature cools over time.

- So: high temperature, accept any *t*
- But, low temperature, behaves like hill-climbing
- Still, $|f(s) - f(t)|$ plays a role: if big, replacement probability low.

Genetic Algorithms

Goal of genetic algorithms: optimize using principles inspired by mechanism for evolution

Analogous to **natural selection, cross-over**, and **mutation**

Break & Quiz

Q 10: Consider hill climbing for a maximization problem, with the following value landscape. Starting from state A with value 5, which neighboring states would standard hill climbing move to?

Neighbors of A: B(value=7), C(value=3), D(value=5), E(value=8):

A. Only state C
B. Only state E
C. States B and E
D. States B, D, and E
E. All neighboring states

Break & Quiz

Q 10: Consider hill climbing for a maximization problem, with the following value landscape. Starting from state A with value 5, which neighboring states would standard hill climbing move to?

Neighbors of A: B(value=7), C(value=3), D(value=5), E(value=8):

A. Only state C
B. Only state E
C. States B and E
D. States B, D, and E
E. All neighboring states

In standard (greedy) hill climbing, the algorithm picks the neighbor with the best value among all neighbors. Since E has value 8, which is the highest among all neighbors (B=7, C=3, D=5, E=8), hill climbing would move to state E. The algorithm doesn't consider multiple states or states with equal/lower value; it greedily selects the single best neighbor.

Thank you and good luck!