Artificial Intelligence

CS 165A

Oct 29, 2020

Instructor: Prof. Yu-Xiang Wang

Today

→ Examples of heuristics in A*-search

→ Games and Adversarial Search

Recap: Search algorithms

- State-space diagram vs Search Tree
- Uninformed Search algorithms
  - BFS / DFS
  - Depth Limited Search
  - Iterative Deepening Search.
  - Uniform cost search.
- Informed Search (with an heuristic function h):
  - Greedy Best-First-Search. (not complete / optimal)
  - A* Search (complete / optimal if h is admissible)

Recap: Summary table of uninformed search

Criteria	BFS	Uniform-cost	DFS	Depth-limited	IDS	Bidirectional
Complete?	Yes$^{\#}$	Yes$^{\#\&}$	No	No	Yes$^{\#}$	Yes$^{\#+}$
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes$^{\$}$	Yes	No	No	Yes$^{\$}$	Yes$^{\$+}$

b: Branching factor
d: Depth of the shallowest goal
l: Depth limit
m: Maximum depth of search tree
e: The lower bound of the step cost

(Section 3.4.7 in the AIMA book.)

$^{\#}$: Complete if b is finite
$^{\&}$: Complete if step cost >= e
$^{\$}$: Optimal if all step costs are identical
$^{+}$: If both direction use BFS

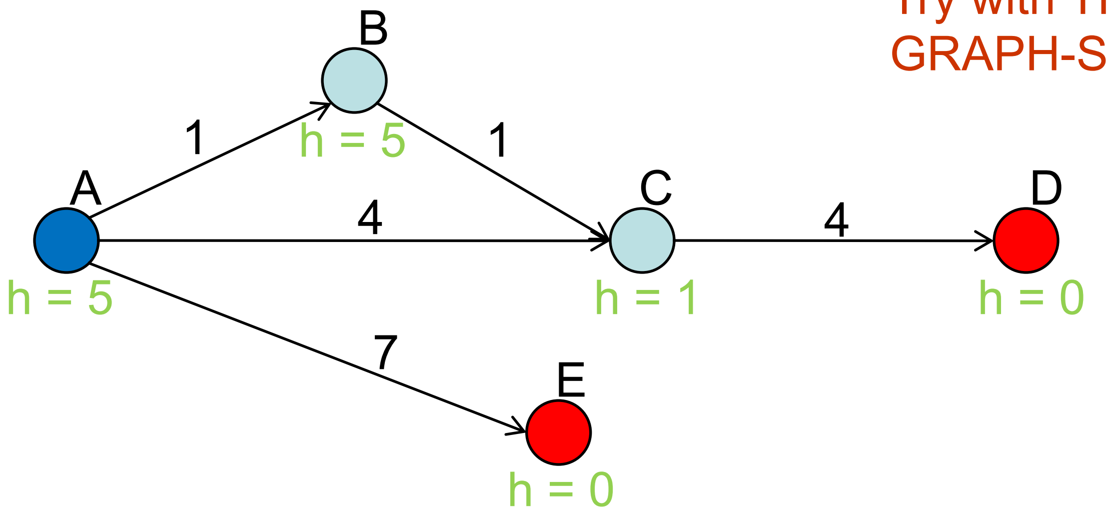
Recap: A* Search (Pronounced "A-Star")

- Uniform-cost search minimizes ***g(n)*** ("past" cost)
- Greedy search minimizes ***h(n)*** ("expected" or "future" cost)
- **"A*** Search" combines the two:
  - Minimize $f(n) = g(n) + h(n)$
  - Accounts for the "past" and the "future"
  - Estimates the cheapest solution (complete path) through node ***n***

function A*-SEARCH(*problem*, ***h***) **returns** a solution or failure
  return BEST-FIRST-SEARCH(*problem*, ***f***)

Recap: Avoiding Repeated States using A* Search

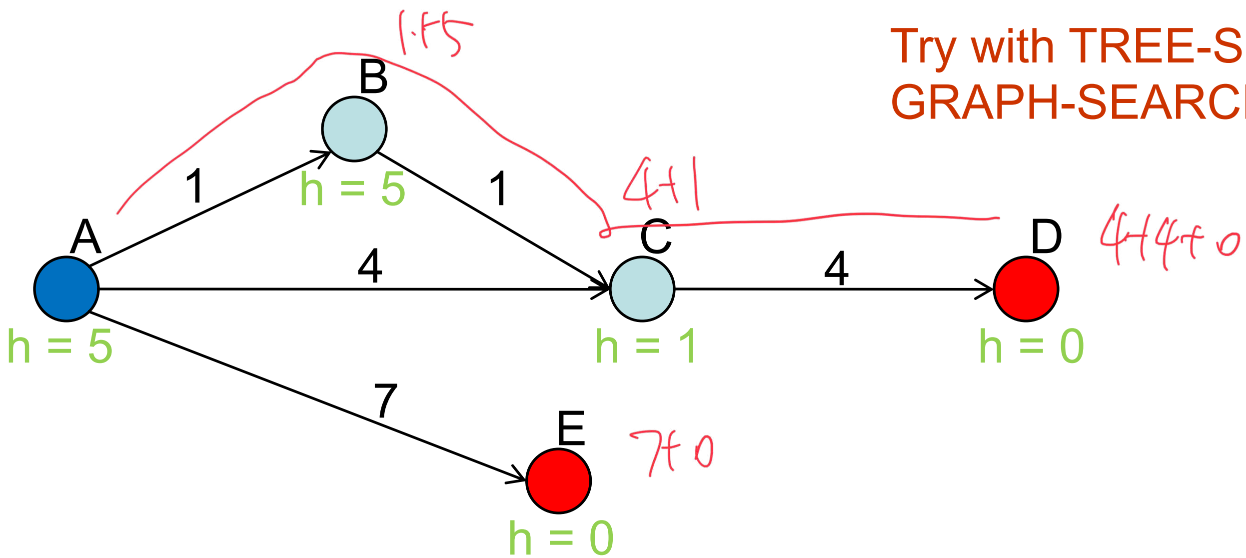
- Is GRAPH-SEARCH optimal with A*?



Try with TREE-SEARCH and GRAPH-SEARCH

Recap: Avoiding Repeated States using A* Search

- Is GRAPH-SEARCH optimal with A*?

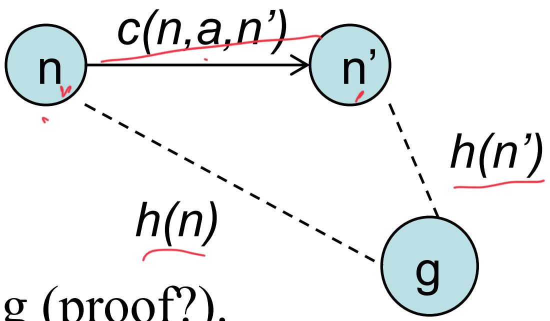


Try with TREE-SEARCH and GRAPH-SEARCH

Graph Search
Step 1: Among B, C, E, Choose C
Step 2: Among B, E, D, Choose B
Step 3: Among D, E, Choose E. (you are not going to select C again) 8 87

Recap: Consistency (Monotonicity) of heuristic h

- A heuristic is consistent (or monotonic) provided
  - for any node n, for any successor n' generated by action a with cost c(n,a,n')
    - $h(n) \leq c(n,a,n') + h(n')$
  - akin to triangle inequality.
  - guarantees admissibility (proof?).
  - values of f(n) along any path are non-decreasing (proof?).
    - Contours of constant f in the state space
- GRAPH-SEARCH using consistent f(n) is optimal.
- Note that h(n) = 0 is consistent and admissible.



This lecture

- Example of heuristics / A* search
  - Effective branching factor

- Games

- Adversarial Search

Heuristics

- What's a heuristic for
  - Driving distance (or time) from city A to city B ? Straight line dist.
  - 8-puzzle problem ? # of misplaced tiles
  - M&C ? # of ppl on the LHS
  - Robot navigation ? # of dirty tiles

Heuristics

- What's a heuristic for
  - Driving distance (or time) from city A to city B ?
  - 8-puzzle problem ?
  - M&C ?
  - Robot navigation ?
- **Admissible** heuristic
  - Does not overestimate the cost to reach the goal
  - "Optimistic"
- **Consistent** heuristic:
  - Satisfy a triangular inequality: $h(n) \leq c(n,a,n') + h(n')$

Heuristics

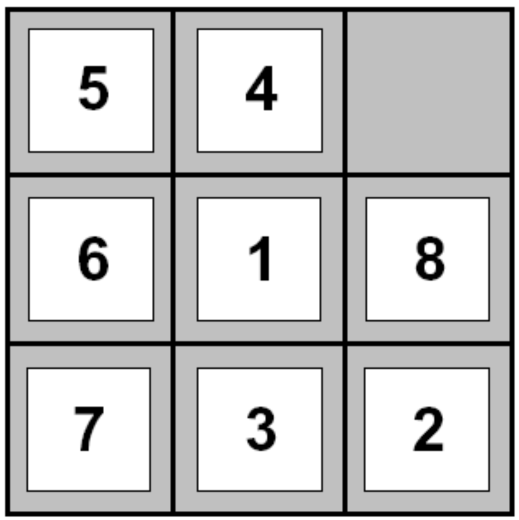
- What's a heuristic for
  - Driving distance (or time) from city A to city B ?
  - 8-puzzle problem ?
  - M&C ?
  - Robot navigation ?
- **Admissible** heuristic
  - Does not overestimate the cost to reach the goal
  - "Optimistic"
- **Consistent** heuristic:
  - Satisfy a triangular inequality: $h(n) \leq c(n,a,n') + h(n')$

$h(n) - h(n') \leq |h(n) - h(n')| \leq 1$

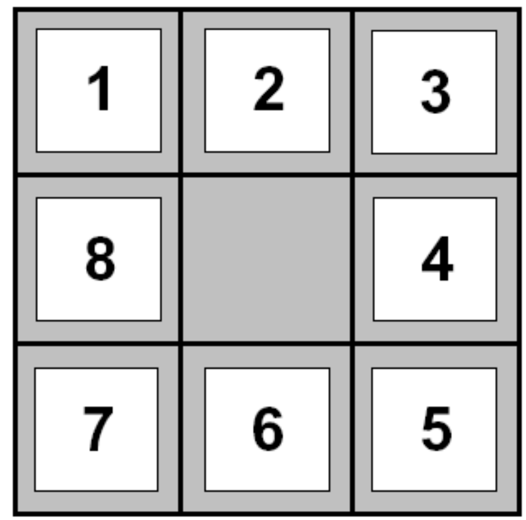
$\therefore h(n) \leq 1 + h(n')$

- Are the above heuristics admissible? Consistent?

Example: 8-Puzzle



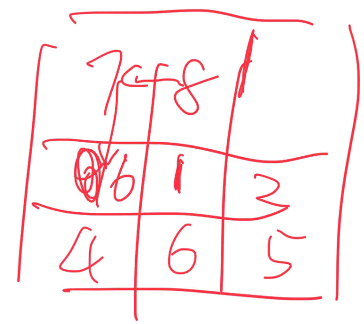
Start State

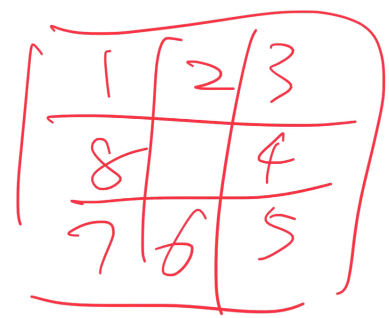


Goal State

Comparing and combining heuristics

- Heuristics generated by considering relaxed versions of a problem.
- Heuristic h_1 for 8-puzzle
  - Number of out-of-order tiles
- Heuristic h_2 for 8-puzzle
  - Sum of Manhattan distances of each tile
- h_2 dominates h_1 provided $h_2(n) \geq h_1(n)$.
  - h_2 will likely prune more than h_1.
- $h =$ $\max(h_1, h_2, .., h_n)$ is
  - admissible if each h_i is
  - consistent if each h_i is
- Cost of sub-problems and pattern databases
  - Cost for 4 specific tiles
  - Can these be added for disjoint sets of tiles?





Effective Branching Factor

Effective Branching Factor

- Though <u>informed</u> search methods may have poor *worst-case* performance, they often do quite well if the heuristic is good
  - Even if there is a huge branching factor

Effective Branching Factor

- Though informed search methods may have poor *worst-case* performance, they often do quite well if the heuristic is good
  - Even if there is a huge branching factor

- One way to quantify the effectiveness of the heuristic: the **effective branching factor, b^***
  - N: total number of nodes expanded
  - d: solution depth
  - $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$

Effective Branching Factor

- Though <u>informed</u> search methods may have poor *worst-case* performance, they often do quite well if the heuristic is good
  - Even if there is a huge branching factor

- One way to quantify the effectiveness of the heuristic: the **effective branching factor, b^***
  - N: total number of nodes expanded
  - d: solution depth
  - $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$

- For a good heuristic, b^* is close to 1

Example: 8-puzzle problem

Averaged over 100 trials each at different solution lengths

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Ave. # of nodes expanded

Solution length

Memory Bounded Search

Memory Bounded Search

- Memory, not computation, is <u>usually</u> the limiting factor in search problems
  - Certainly true for A* search

Memory Bounded Search

- Memory, not computation, is <u>usually</u> the limiting factor in search problems
  - Certainly true for A* search
- Why? What takes up memory in A* search?

Memory Bounded Search

- Memory, not computation, is usually the limiting factor in search problems
  - Certainly true for A* search
- Why? What takes up memory in A* search?

- Solution: Memory-bounded A* search
  - Iterative Deepening A* (IDA*)
  - Simplified Memory-bounded A* (SMA*) Beam Search
  - (Read the textbook for more details.)

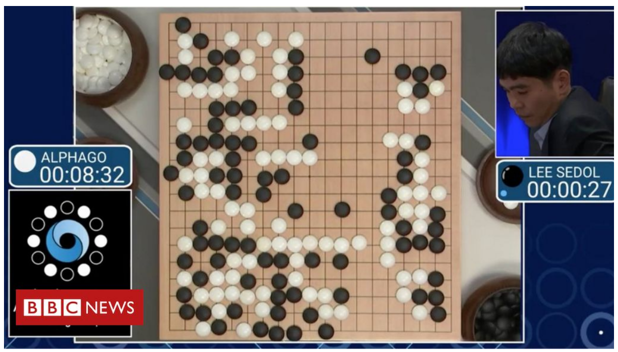
Summary of informed search

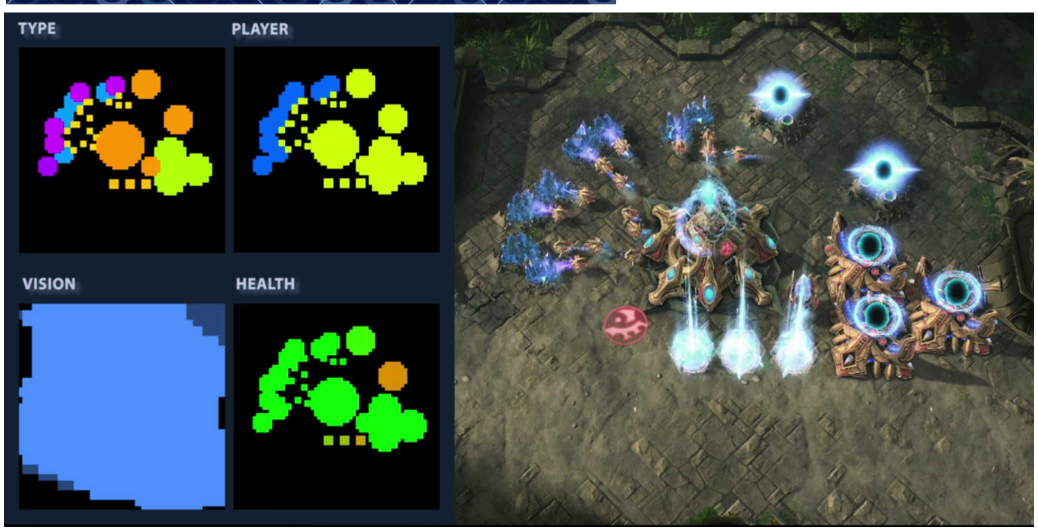
- How to use a heuristic function to improve search
  - Greedy Best-first search + Uniform-cost search = A* Search

- When is A* search optimal?
  - h is Admissible (optimistic) for Tree Search
  - h is Consistent for Graph Search

- Choosing heuristic functions
  - A good heuristic function can reduce time/space cost of search by orders of magnitude.
  - Good heuristic function may take longer to evaluate.

Games and Adversarial Search





- Games: problem setup
- Minimax search
- Alpha-beta pruning

Illustrative example of a simple game (1 min discussion)

Example: game 1

You choose one of the three bins.
I choose a number from that bin.
Your goal is to maximize the chosen number.

A		B		C	
-50	50	1	3	-5	15

$E[r]$
$= 15 \cdot 0.5$
$+ (-5) \cdot 0.5$

(Example taken from Liang and Sadigh)

Game as a search problem

Game as a search problem

- S_0 The initial state

Game as a search problem

- S_0 The initial state

- PLAYER(s): Returns which player has the move

Game as a search problem

- S_0 The initial state
- PLAYER(s): Returns which player has the move
- ACTIONS(s): Returns the legal moves.

Game as a search problem

- S_0 The initial state
- PLAYER(s): Returns which player has the move
- ACTIONS(s): Returns the legal moves.
- RESULT(s, a): Output the state we transition to.

Game as a search problem

- S_0 The initial state
- PLAYER(s): Returns which player has the move
- ACTIONS(s): Returns the legal moves.
- RESULT(s, a): Output the state we transition to.
- TERMINAL-TEST(s): Returns True if the game is over.

Game as a search problem

- S_0 The initial state
- PLAYER(s): Returns which player has the move
- ACTIONS(s): Returns the legal moves.
- RESULT(s, a): Output the state we transition to.
- TERMINAL-TEST(s): Returns True if the game is over.
- UTILITY(s,p): The payoff of player p at terminal state s.

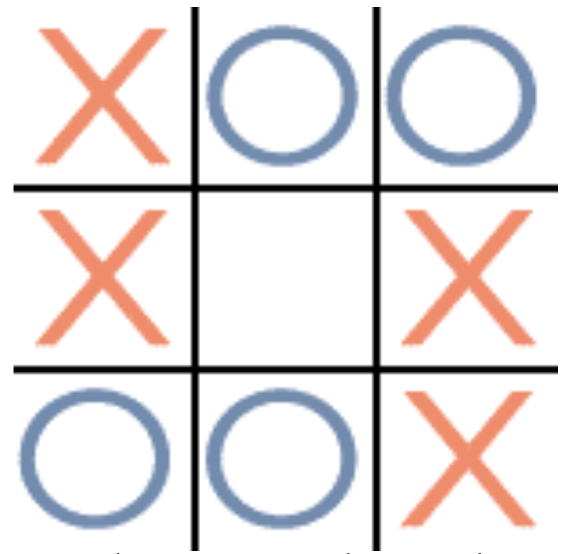
[illegible]

$$\begin{array}{r} 01 \\ +\ 10 \\ \hline 11 \end{array}$$

Two-player, Turn-based, Perfect information, Deterministic, Zero-Sum Game

- Two-player: Tic-Tac-Toe, Chess, Go!
- Turn-based: The players take turns in round-robin fashion.
- Perfect information: The State is known to everyone
- Deterministic: Nothing is random
- Zero-sum: The total payoff for all players is a constant.
  - *The 8-puzzle is a one-player, perfect info, deterministic, zero-sum game.*
  - *How about Rock-Paper-Scissors?*
  - *How about Monopoly?*
  - *How about Starcraft?*

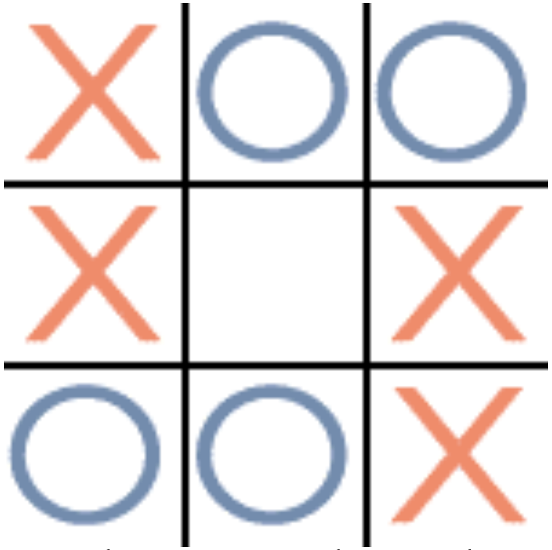
Tic-Tac-Toe



- The first player is **X** and the second is **O**
- Object of game: get three of your symbol in a horizontal, vertical or diagonal row on a 3x3 game board
- **X** always goes first
- Players alternate placing **X**s and **O**s on the game board
- Game ends when a player has three in a row (a wins) or all nine squares are filled (a draw)

Tic-Tac-Toe

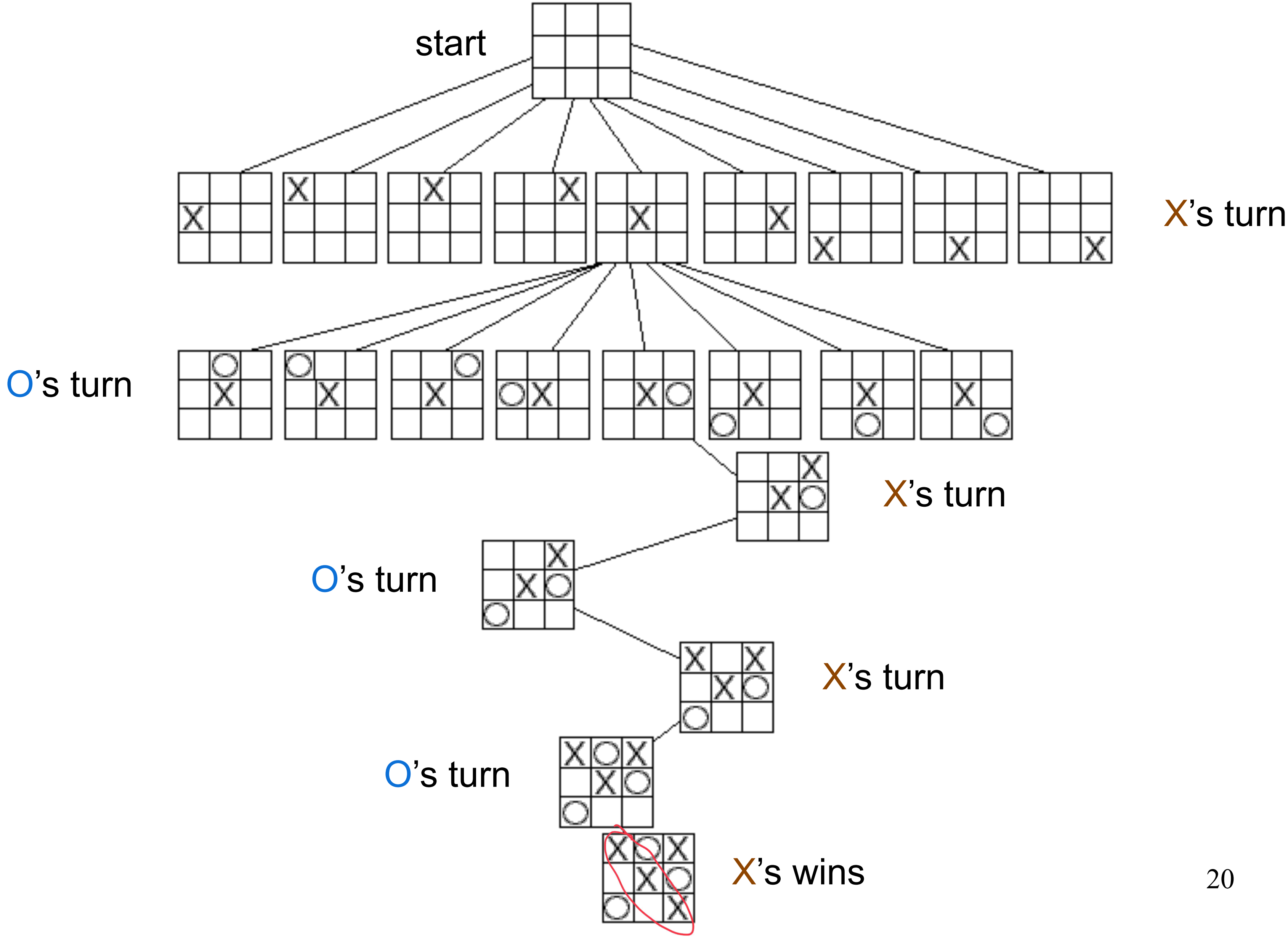
- The first player is **X** and the second is **O**
- Object of game: get three of your symbol in a horizontal, vertical or diagonal row on a 3x3 game board
- **X** always goes first



- Players alternate placing **X**s and **O**s on the game board
- Game ends when a player has three in a row (a wins) or all nine squares are filled (a draw)

What's the state, action, transition, payoff for Tic-Tac-Toe?

Partial game tree for Tic-Tac-Toe



Game trees

- A game tree is like a search tree in many ways …
  - nodes are search states, with full details about a position
    - characterize the arrangement of game pieces on the game board
  - edges between nodes correspond to moves
  - leaf nodes correspond to a set of goals
    - { win, lose, draw }
    - usually determined by a score for or against player
  - at each node it is one or other player's turn to move
- A game tree is not like a search tree because you have an opponent!

Two players: MIN and MAX

- In a zero-sum game:
  - payoff to Player 1 = - payoff to Player 2

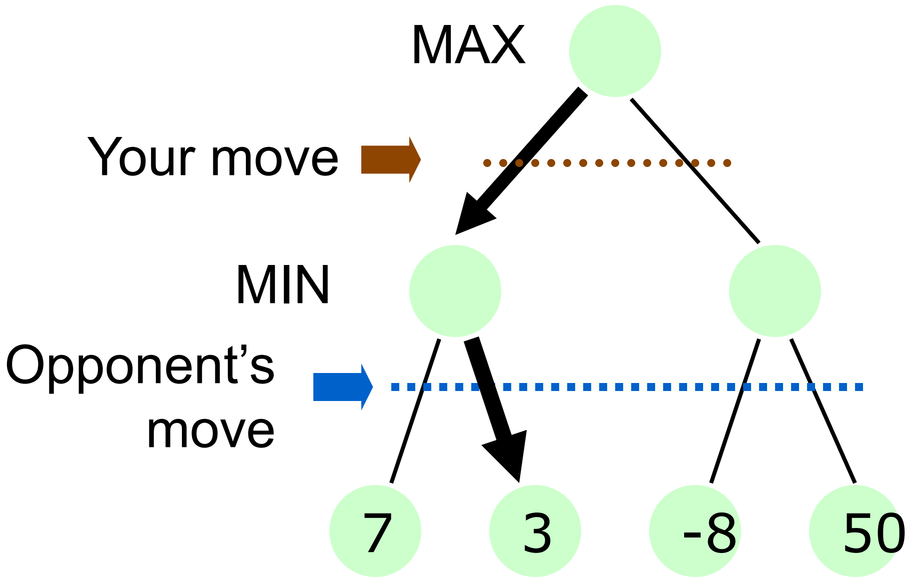
- The goal of Player 1 is to maximizing her payoff.

- The goal of Player 2 is to maximizing her payoff as well
  - Equivalent to minimizing Player 1's payoff.

Minimax search

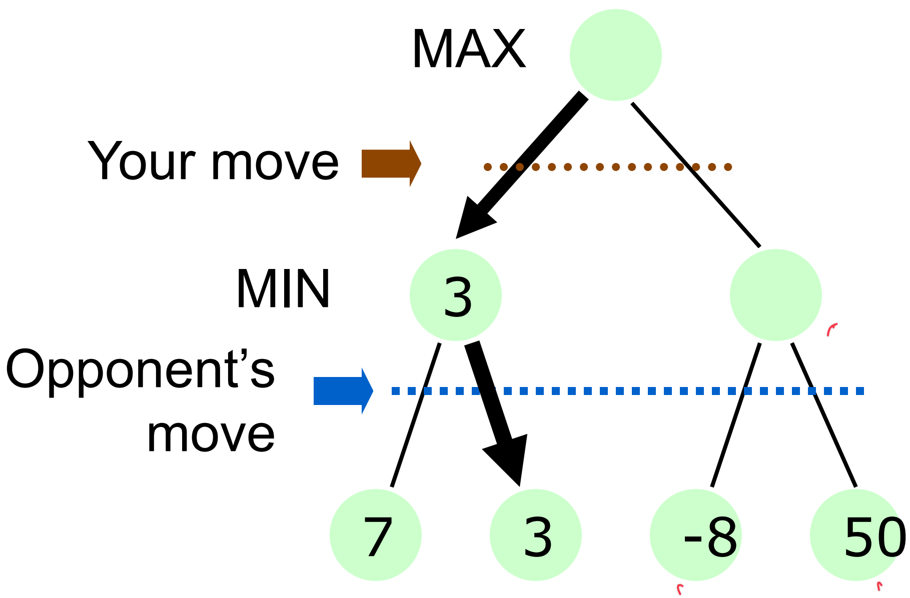
- Assume that both players play perfectly
  - do not assume player will miss good moves or make mistakes
- Score(s): The score that MAX will get towards the end if both player play perfectly from s onwards.
- Consider MIN's strategy
  - MIN's best strategy:
    - choose the move that minimizes the score that will result when MAX chooses the maximizing move
  - MAX does the opposite

Minimaxing



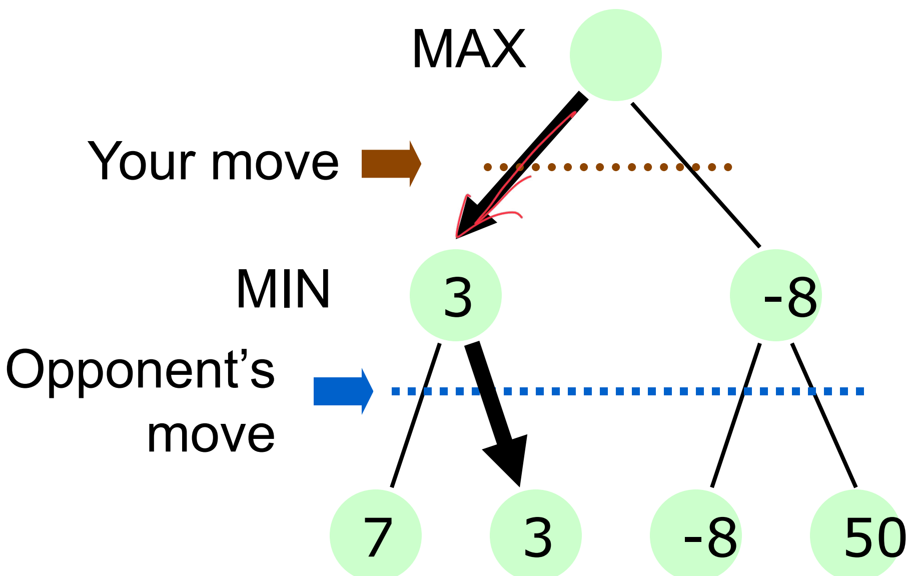
- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Thus your choices are only 3 or -8
- You should move left

Minimaxing



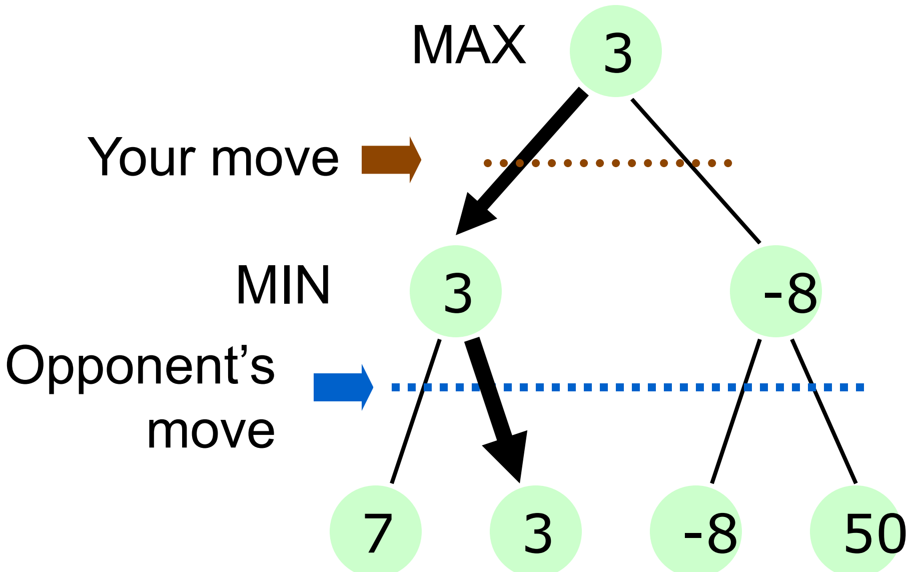
- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Thus your choices are only 3 or -8
- You should move left

Minimaxing



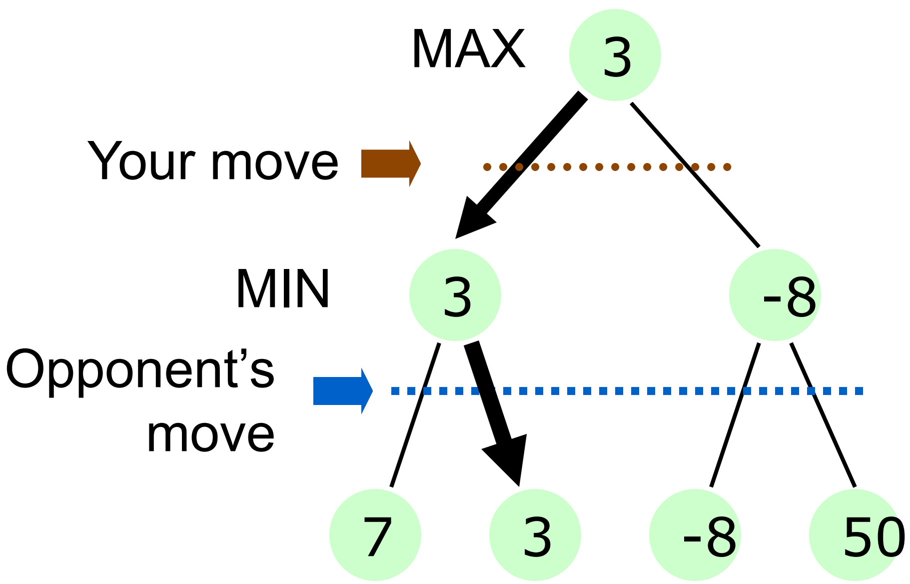
- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Thus your choices are only 3 or -8
- You should move left

Minimaxing



- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Thus your choices are only 3 or -8
- You should move left

Minimaxing

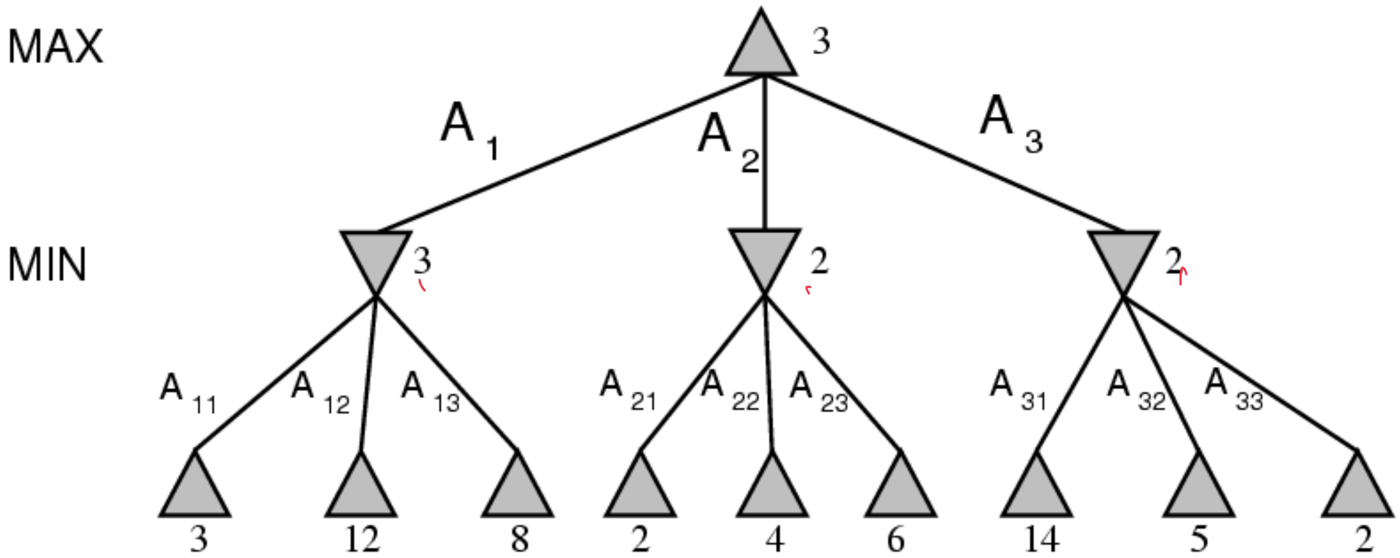


- Your opponent will choose smaller numbers
- If you move left, your opponent will choose 3
- If you move right, your opponent will choose -8
- Thus your choices are only 3 or -8
- You should move left

Each move is called a "ply". One round is K-plies for a K-player game.

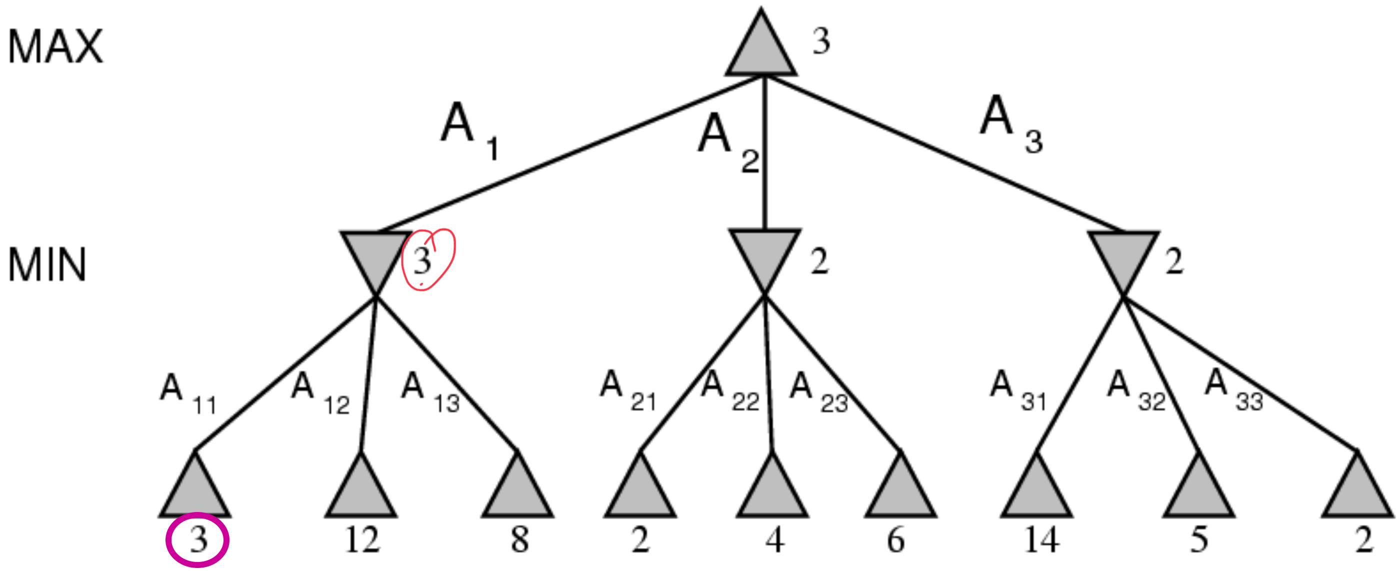
Minimax example

Which move to choose?



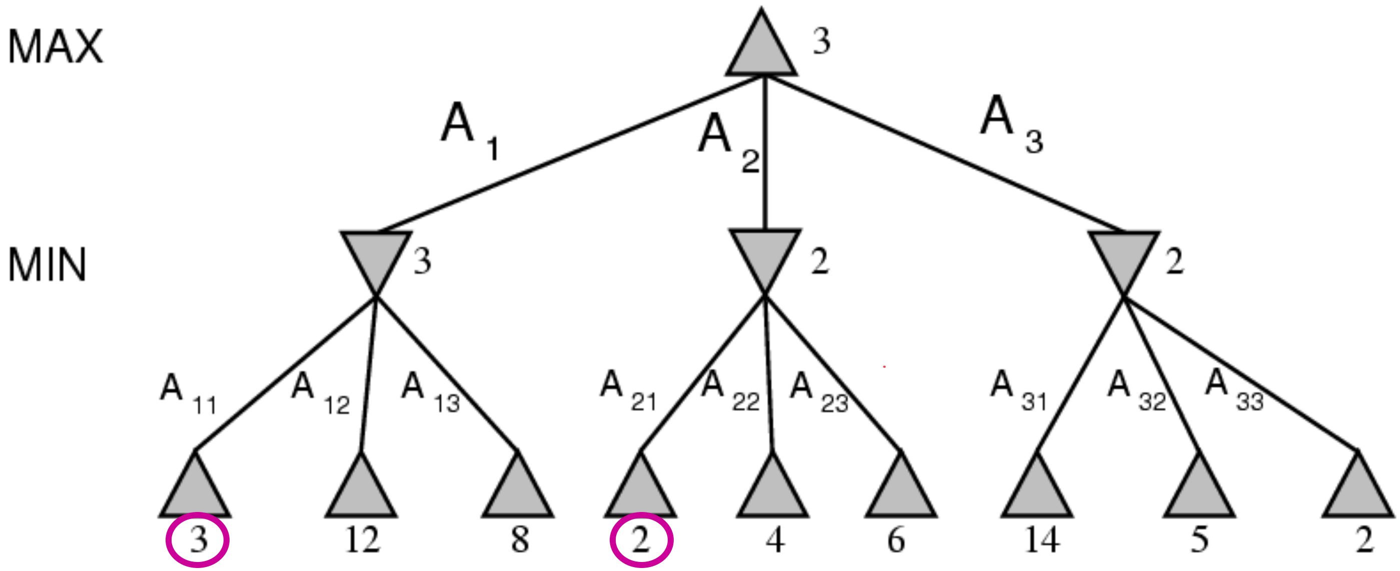
Minimax example

Which move to choose?



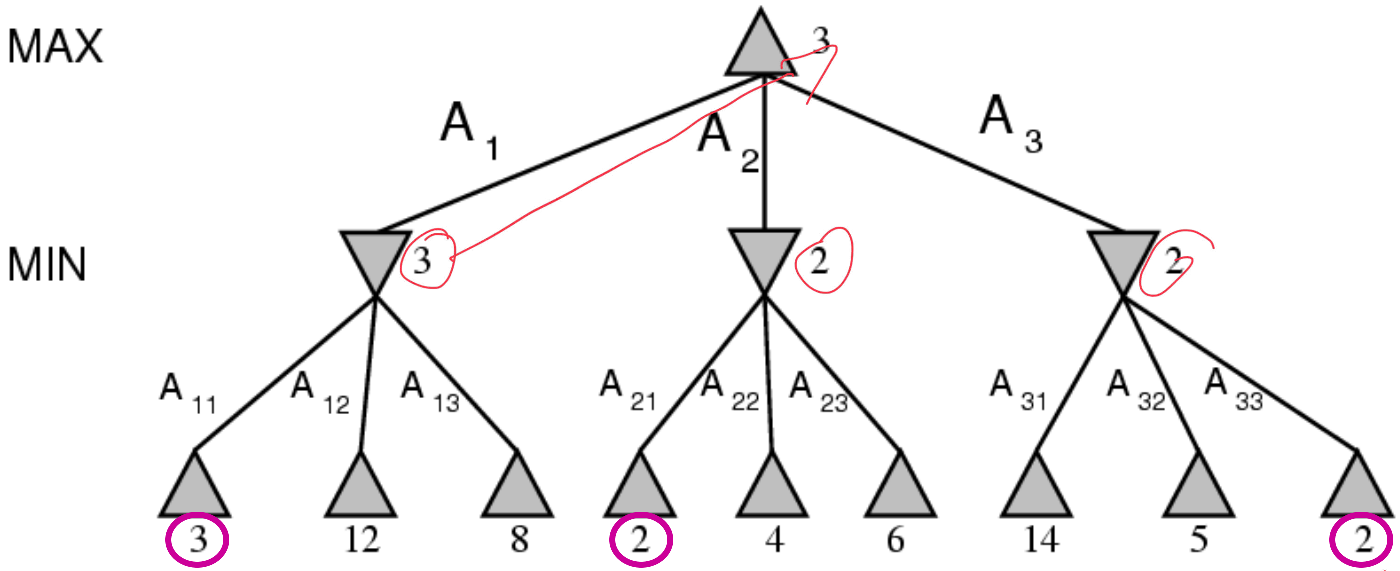
Minimax example

Which move to choose?



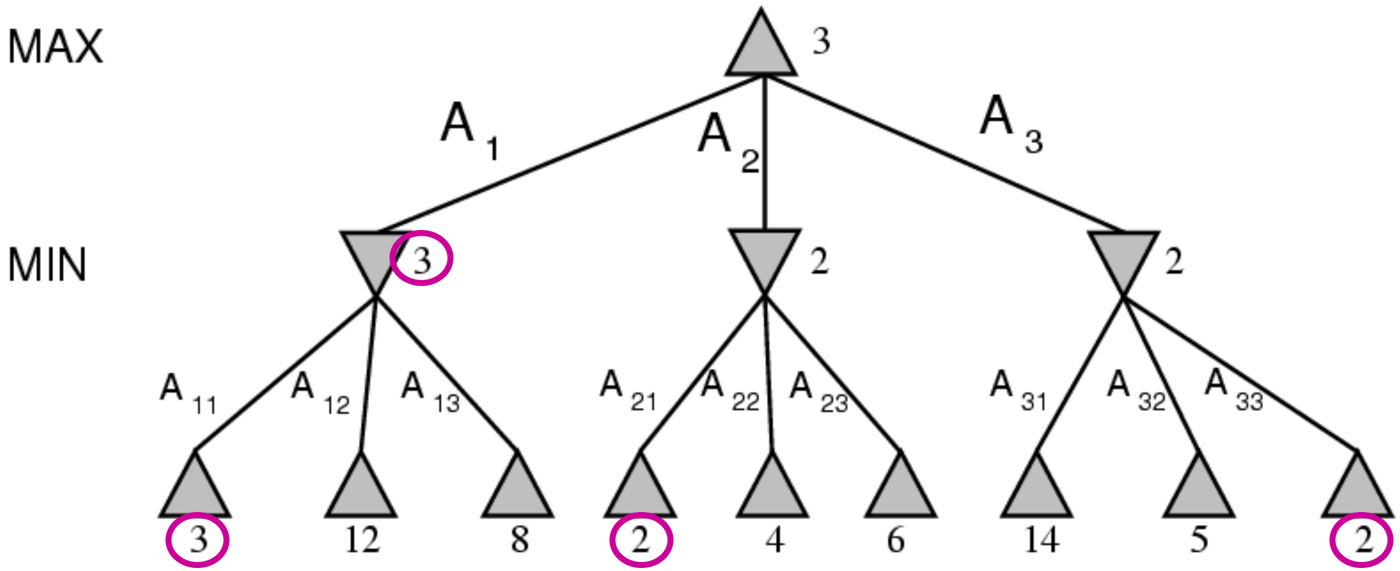
Minimax example

Which move to choose?



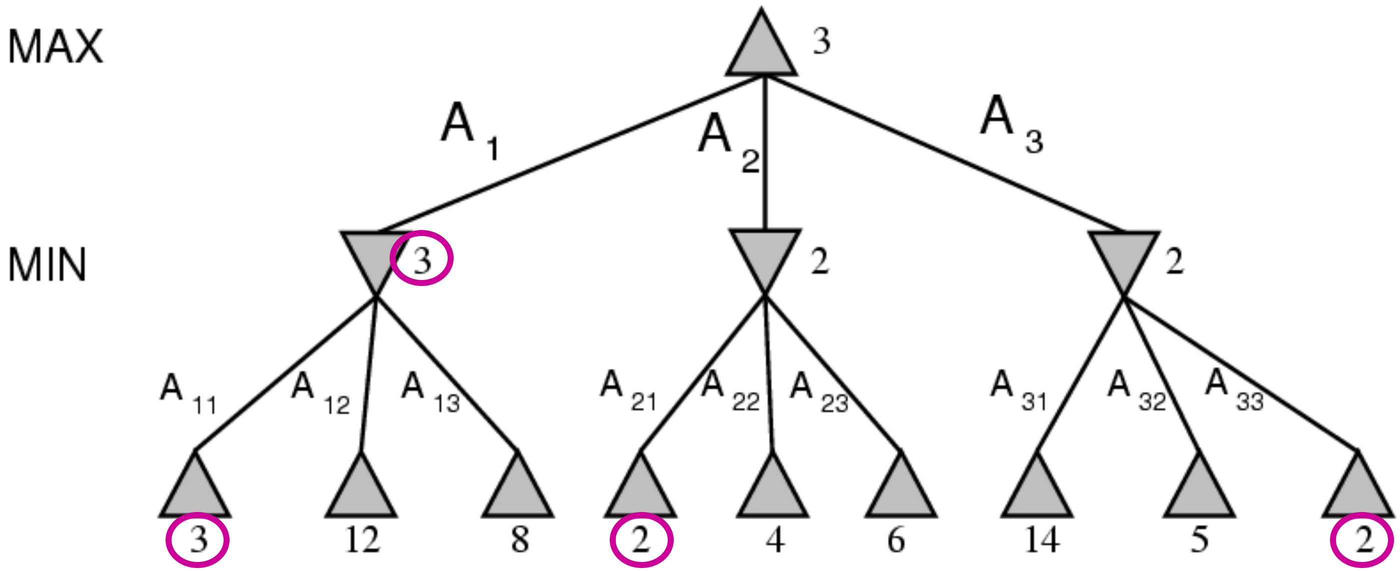
Minimax example

Which move to choose?



Minimax example

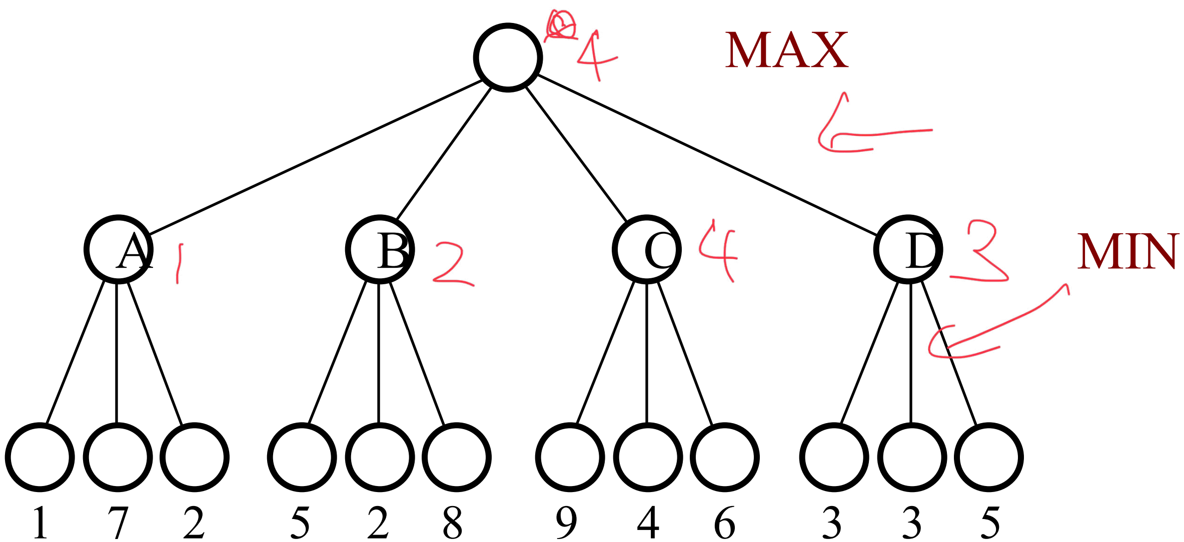
Which move to choose?



The **minimax decision** is move $\mathbf{A_1}$

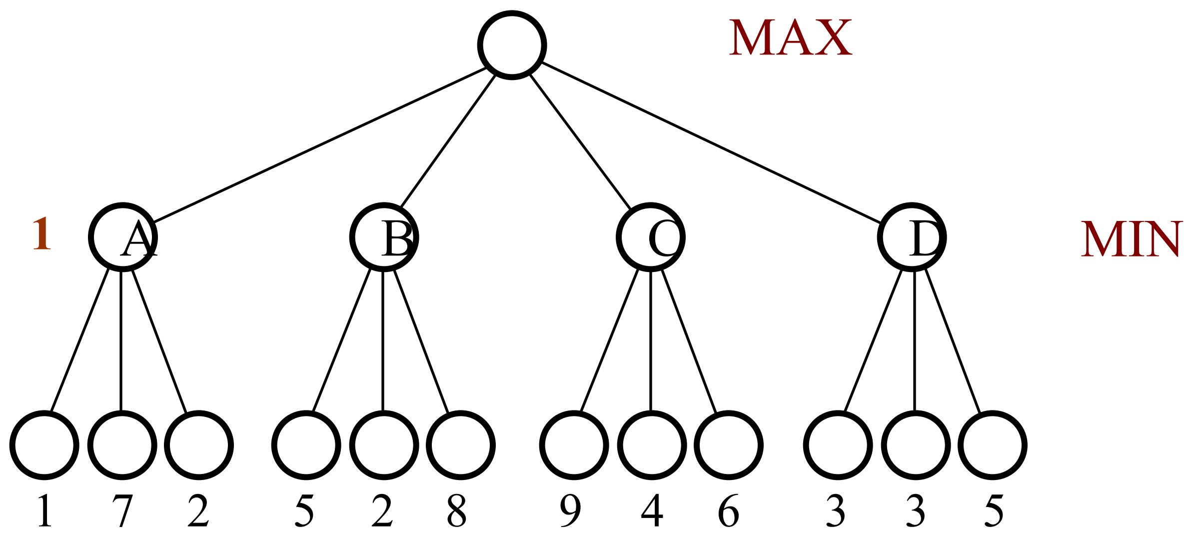
Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



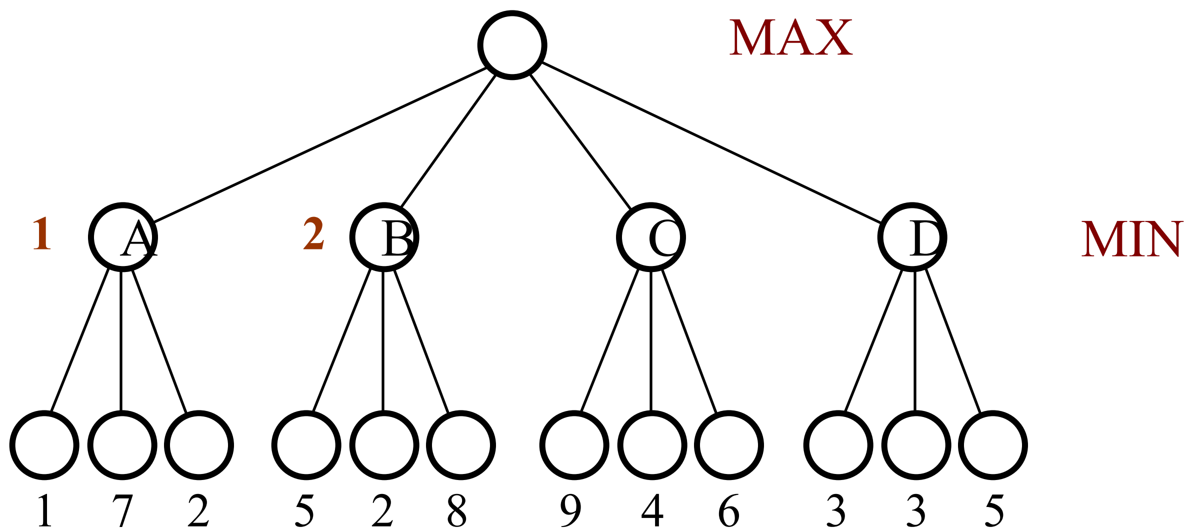
Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



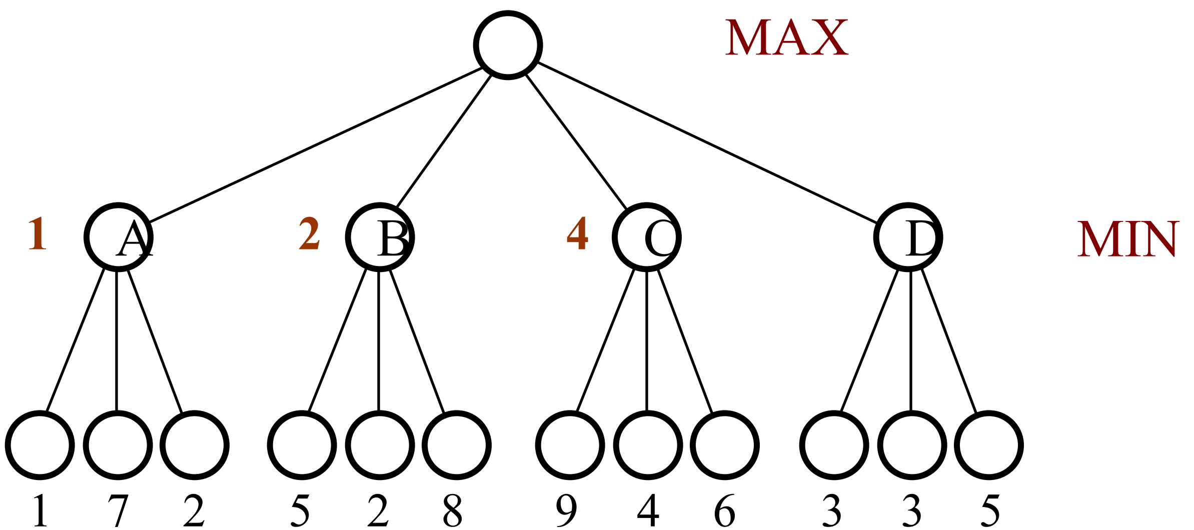
Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



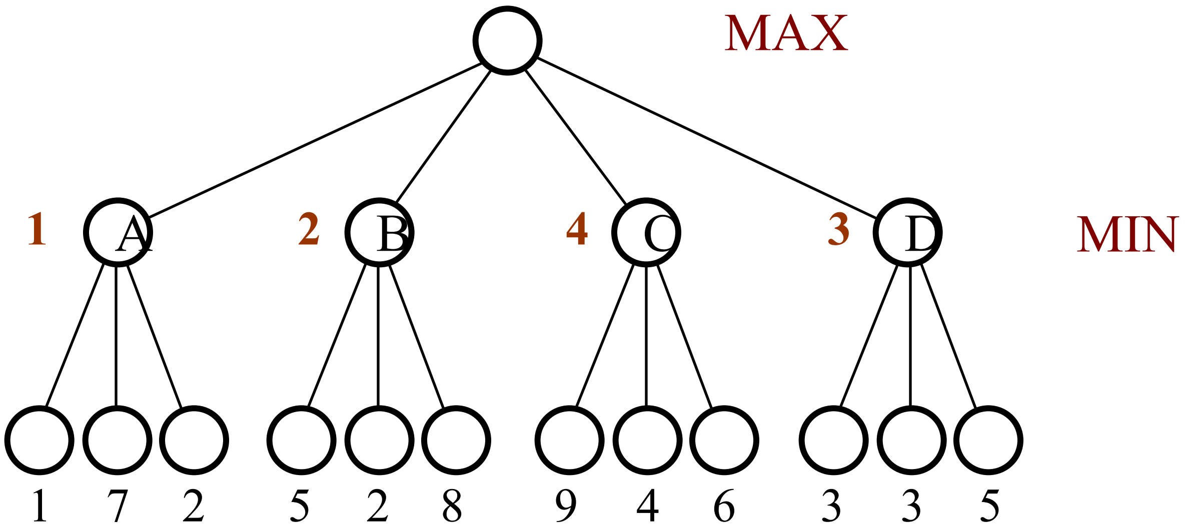
Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



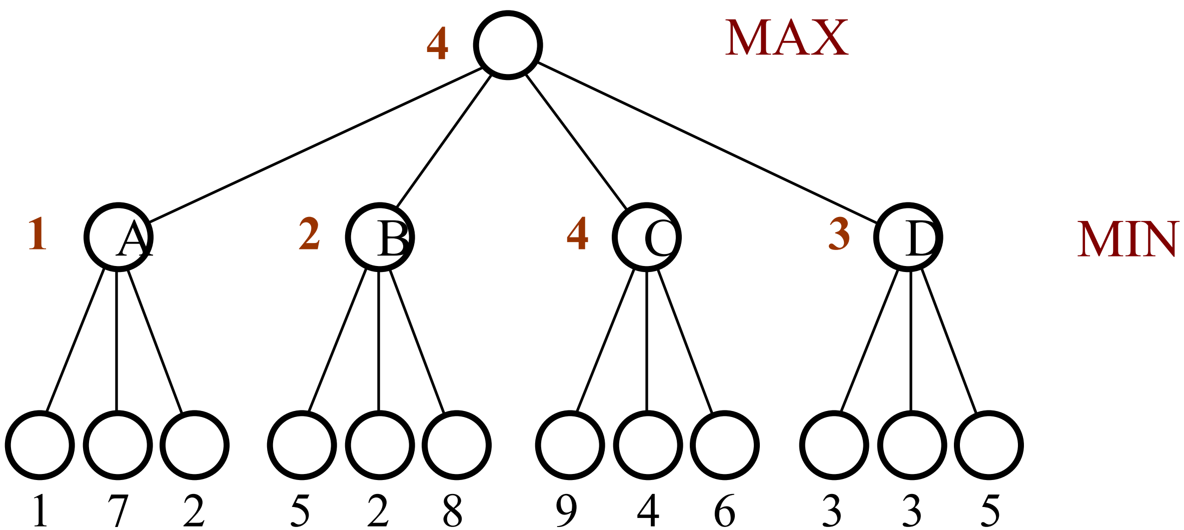
Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



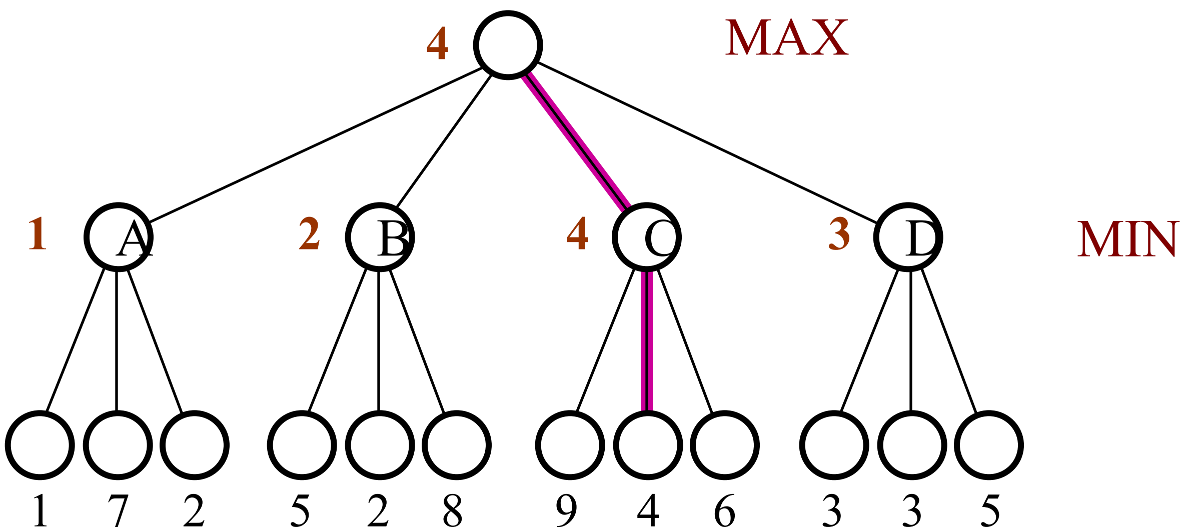
Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



Another example

- In the game, it's your move. Which move will the minimax algorithm choose – A, B, C, or D? What is the minimax value of the root node and nodes A, B, C, and D?



Minimax search

Minimax search

- The *minimax decision* maximizes the utility under the assumption that the opponent seeks to minimize it (if it uses the same evaluation function)

Minimax search

- The *minimax decision* maximizes the utility under the assumption that the opponent seeks to minimize it (if it uses the same evaluation function)

- Generate the tree of minimax values
  - Then choose best (maximum) move
  - Don't need to keep all values around
    - Good memory property

Minimax search

- The *minimax decision* maximizes the utility under the assumption that the opponent seeks to minimize it (if it uses the same evaluation function)

- Generate the tree of minimax values
  - Then choose best (maximum) move
  - Don't need to keep all values around
    - Good memory property

- Depth-first search is used to implement minimax
  - Expand all the way down to leaf nodes
  - Recursive implementation

Minimax properties

- Optimal?

  Yes, against an optimal opponent, **if** the tree is finite

- Complete?

  Yes, **if** the tree is finite

- Time complexity?

  Exponential: **O(** $\boldsymbol{b^m}$ **)**

- Space complexity?

  Polynomial: **O(** $\boldsymbol{bm}$ **)**

But this could take forever…

- Exact search is intractable
  - Tic-Tac-Toe is 9! = 362,880
  - For chess, $b \approx 35$ and $m \approx 100$ for "reasonable" games
  - Go is 361! $\approx 10^{750}$

But this could take forever…

- Exact search is intractable
  - Tic-Tac-Toe is 9! = 362,880
  - For chess, $b \approx 35$ and $m \approx 100$ for “reasonable” games
  - Go is 361! $\approx 10^{750}$

- Idea 1: Pruning

But this could take forever…

- Exact search is intractable
  - Tic-Tac-Toe is 9! = 362,880
  - For chess, $b \approx 35$ and $m \approx 100$ for "reasonable" games
  - Go is 361! $\approx 10^{750}$

- Idea 1: Pruning

- Idea 2: Cut off early and use a heuristic function

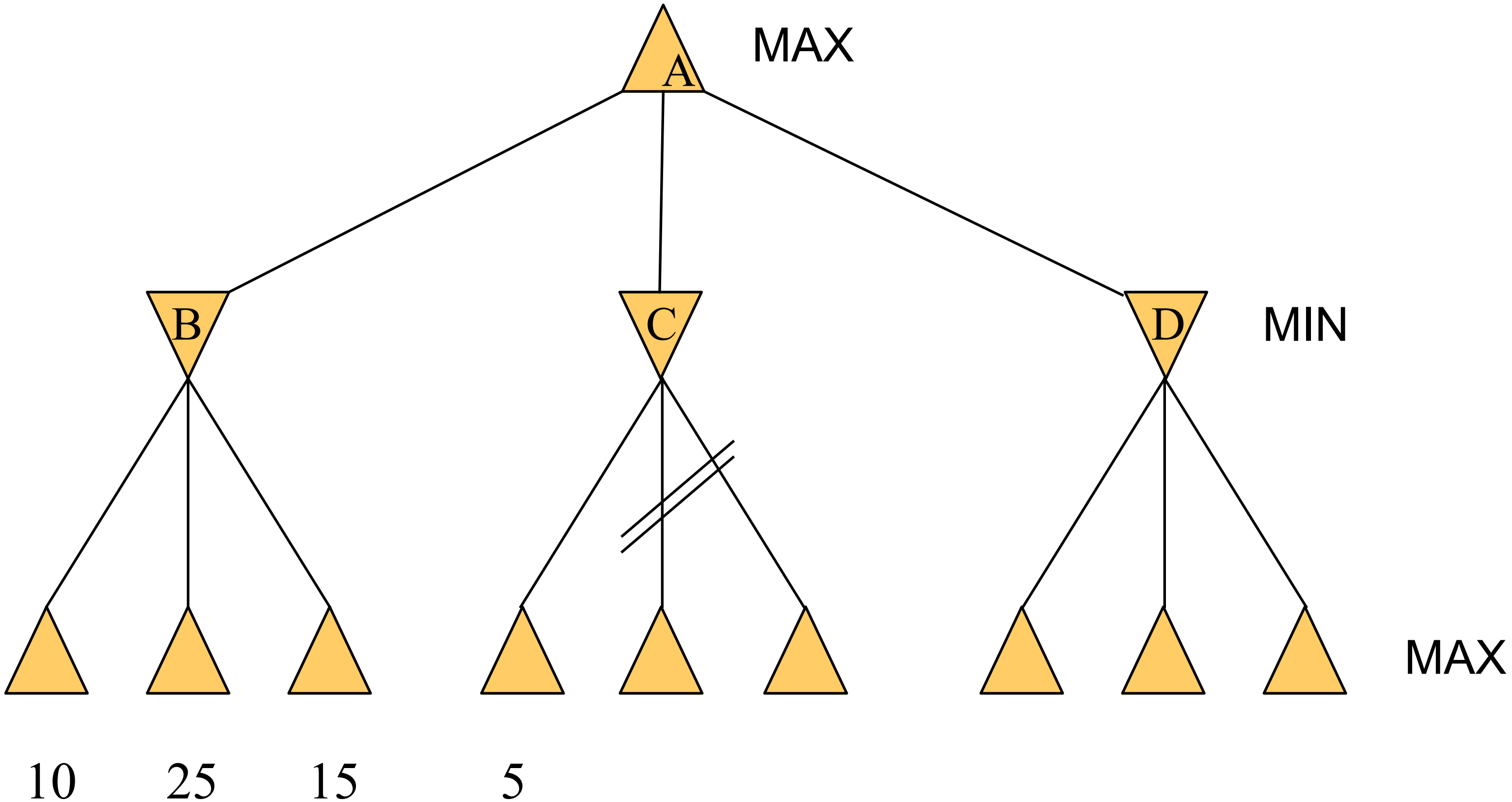
Pruning

- What's really needed is "smarter," more efficient search
  - Don't expand "dead-end" nodes!
- **Pruning** – eliminating a branch of the search tree from consideration

Pruning

- What's really needed is "smarter," more efficient search
  - Don't expand "dead-end" nodes!
- **Pruning** – eliminating a branch of the search tree from consideration
- **Alpha-beta pruning**, applied to a minimax tree, returns the same "best" move, while pruning away unnecessary branches
  - Many fewer nodes might be expanded
  - Hence, smaller effective branching factor
  - …and deeper search
  - …and better performance
    - Remember, minimax is *depth-first* search

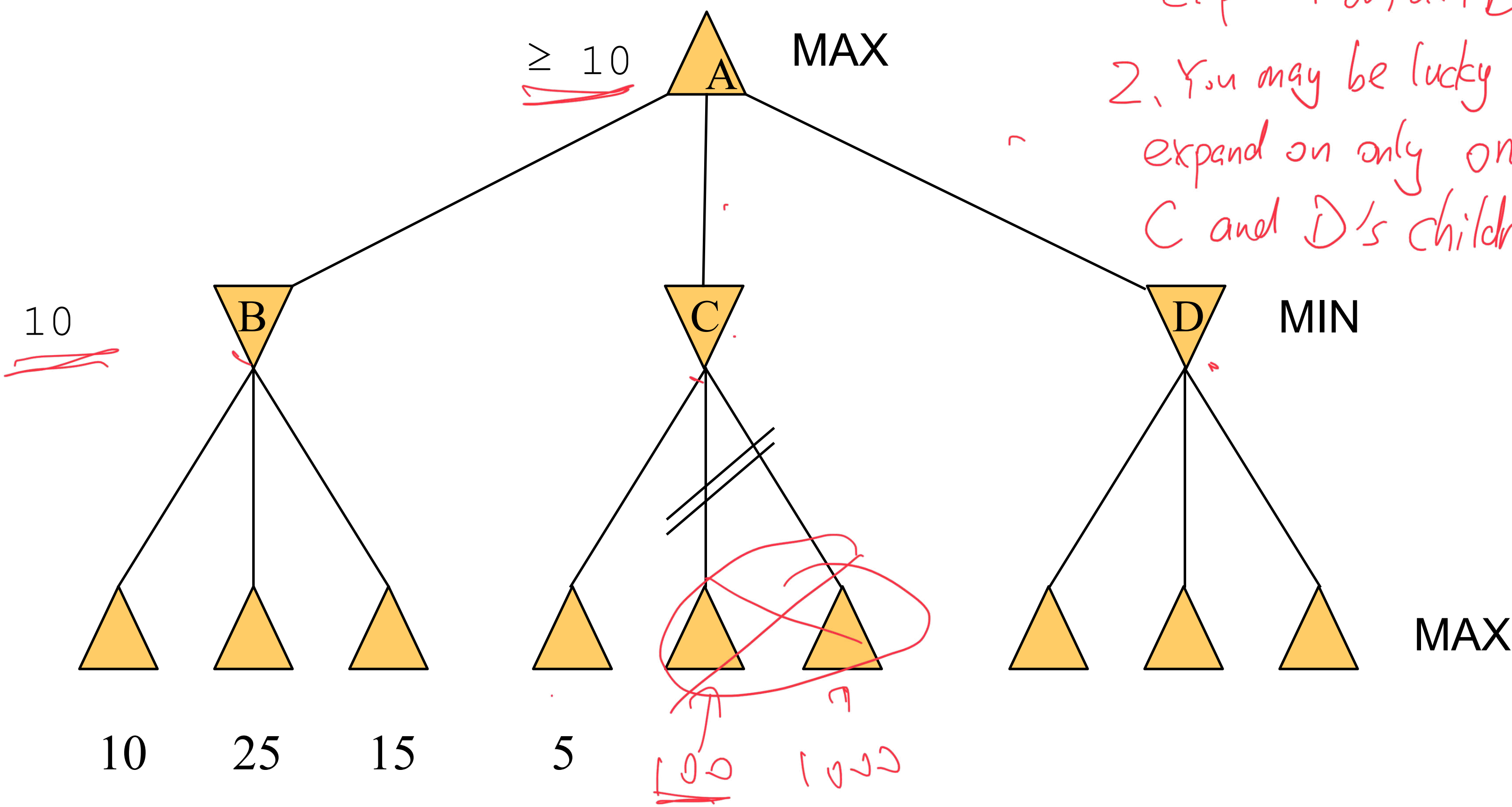
Alpha pruning



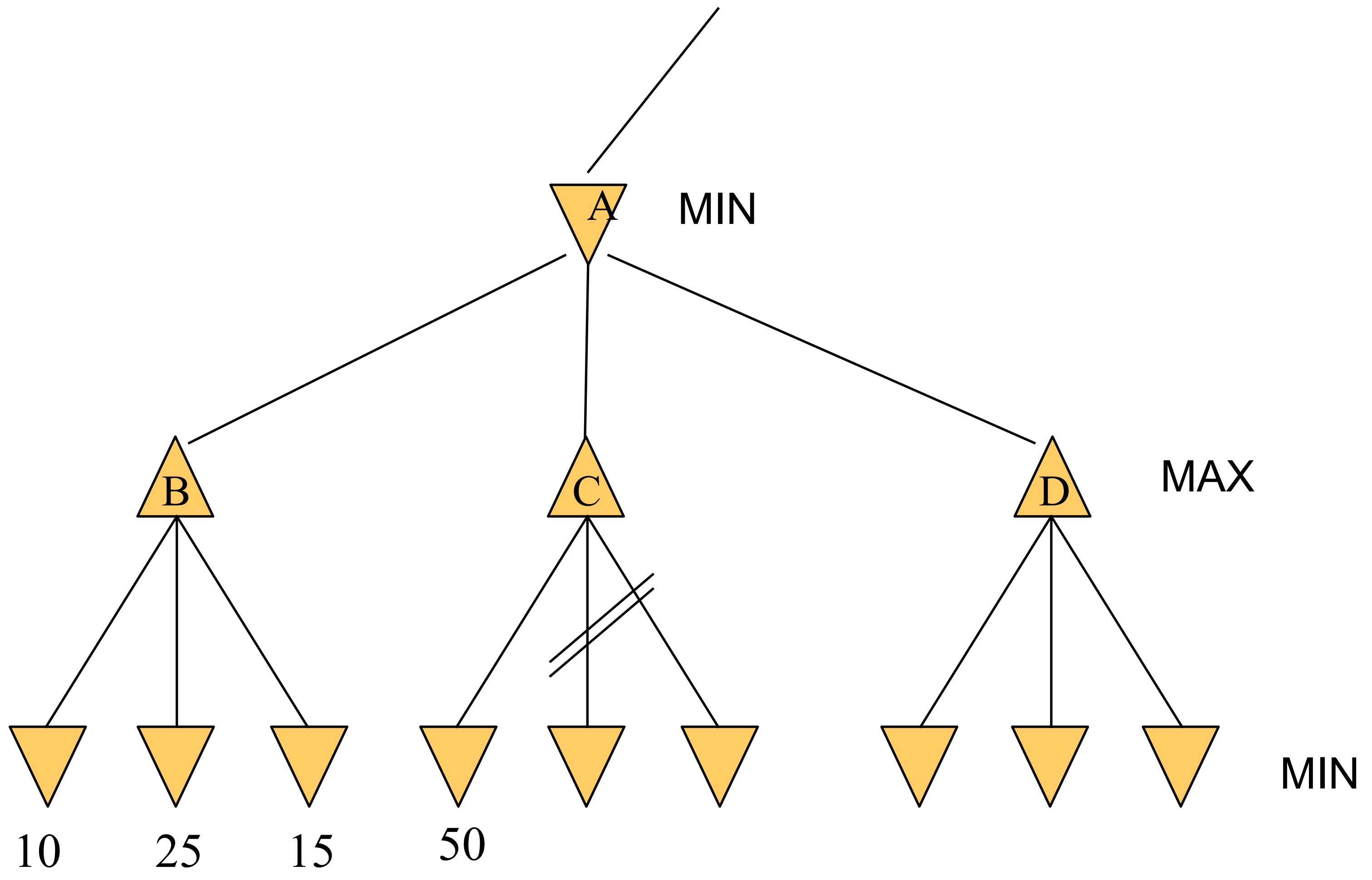
Alpha pruning

Notes:

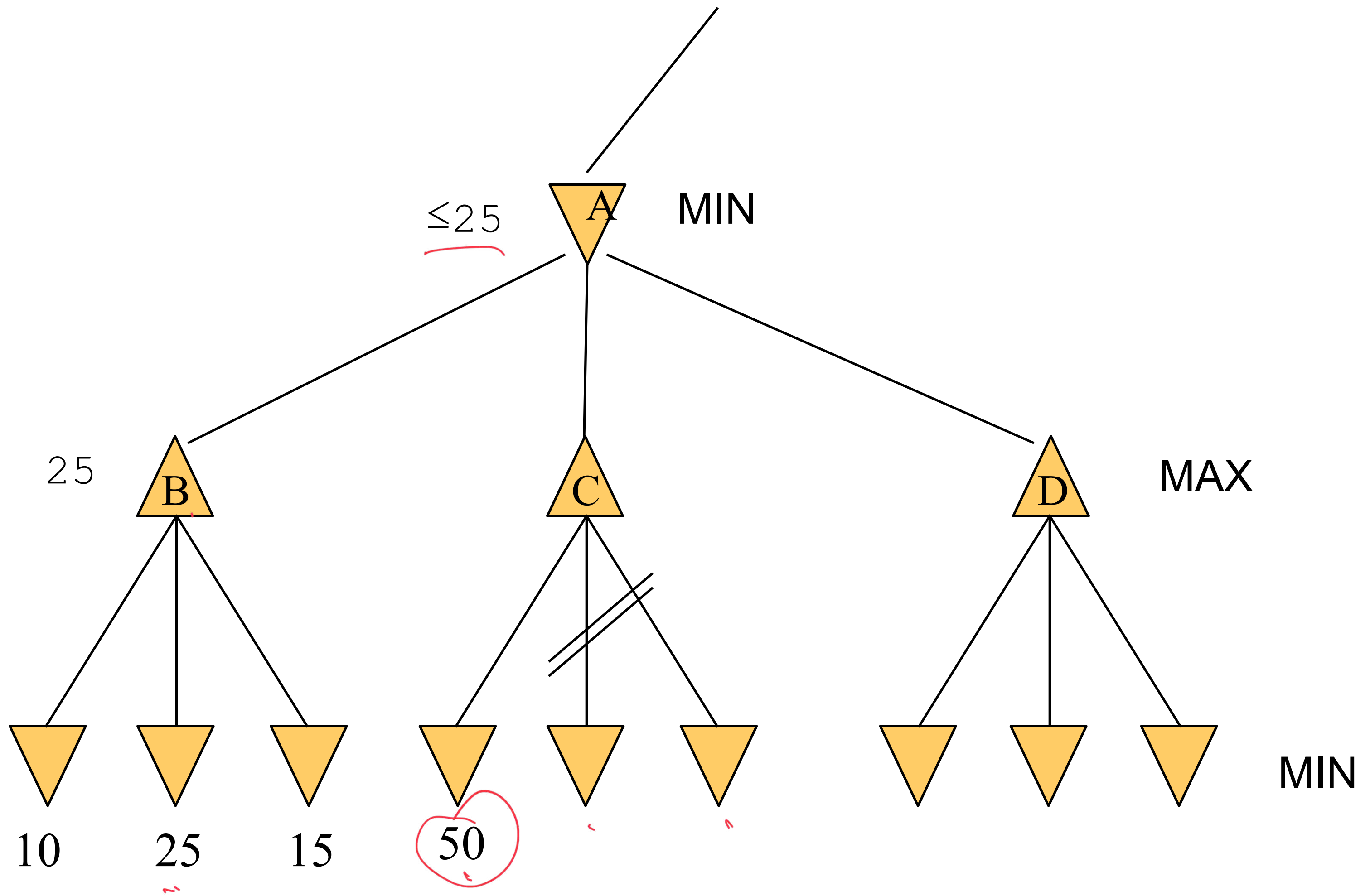
1. You still have to expand on all B, C, D
2. You may be lucky and expand on only one of C and D's children.



Beta pruning



Beta pruning



Improvements via alpha/beta pruning

Improvements via alpha/beta pruning

- Depends on the ordering of expansion

Improvements via alpha/beta pruning

- Depends on the ordering of expansion
- Perfect ordering $O(b^{m/2})$

Improvements via alpha/beta pruning

- Depends on the ordering of expansion
- Perfect ordering $O(b^{m/2})$
- Random ordering $O(b^{3m/4})$

Improvements via alpha/beta pruning

- Depends on the ordering of expansion

- Perfect ordering $O(b^{m/2})$

- Random ordering $O(b^{3m/4})$

- For specific games like Chess, you can get to almost perfect ordering.

Heuristic (Evaluation function)

- It is usually impossible to solve games completely
- Rather, cut the search off early and apply a heuristic evaluation function to the leaves
  - ***h*(*s*)** estimates the expected utility of the game from a given position (node/state) ***s***
  - like depth bounded depth first, lose completeness
  - Explore game tree using combination of evaluation function and search

Heuristic (Evaluation function)

- It is usually impossible to solve games completely
- Rather, cut the search off early and apply a heuristic evaluation function to the leaves
  - ***h*(*s*)** estimates the expected utility of the game from a given position (node/state) ***s***
  - like depth bounded depth first, lose completeness
  - Explore game tree using combination of evaluation function and search
- The performance of a game-playing program depends on the quality (and speed!) of its evaluation function

Heuristics (Evaluation function)

- Typical evaluation function for game: weighted linear function
  - $h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_d f_d(s)$
  - *weights* · *features* [dot product]

Heuristics (Evaluation function)

- Typical evaluation function for game: weighted linear function
  - $h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_d f_d(s)$
  - *weights* · *features* [dot product] $\vec{w}^T \vec{f}(s)$
- For example, in chess
  - $W = \{ 1, 3, 3, 5, 8 \}$
  - $F = \{$ # pawns advantage, # bishops advantage, # knights advantage, # rooks advantage, # queens advantage $\}$

Heuristics (Evaluation function)

- Typical evaluation function for game: weighted linear function
  - $h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_d f_d(s)$
  - *weights* · *features* [dot product]
- For example, in chess
  - $W = \{ 1, 3, 3, 5, 8 \}$
  - $F = \{$ # pawns advantage, # bishops advantage, # knights advantage, # rooks advantage, # queens advantage $\}$
  - Is this what Deep Blue used?

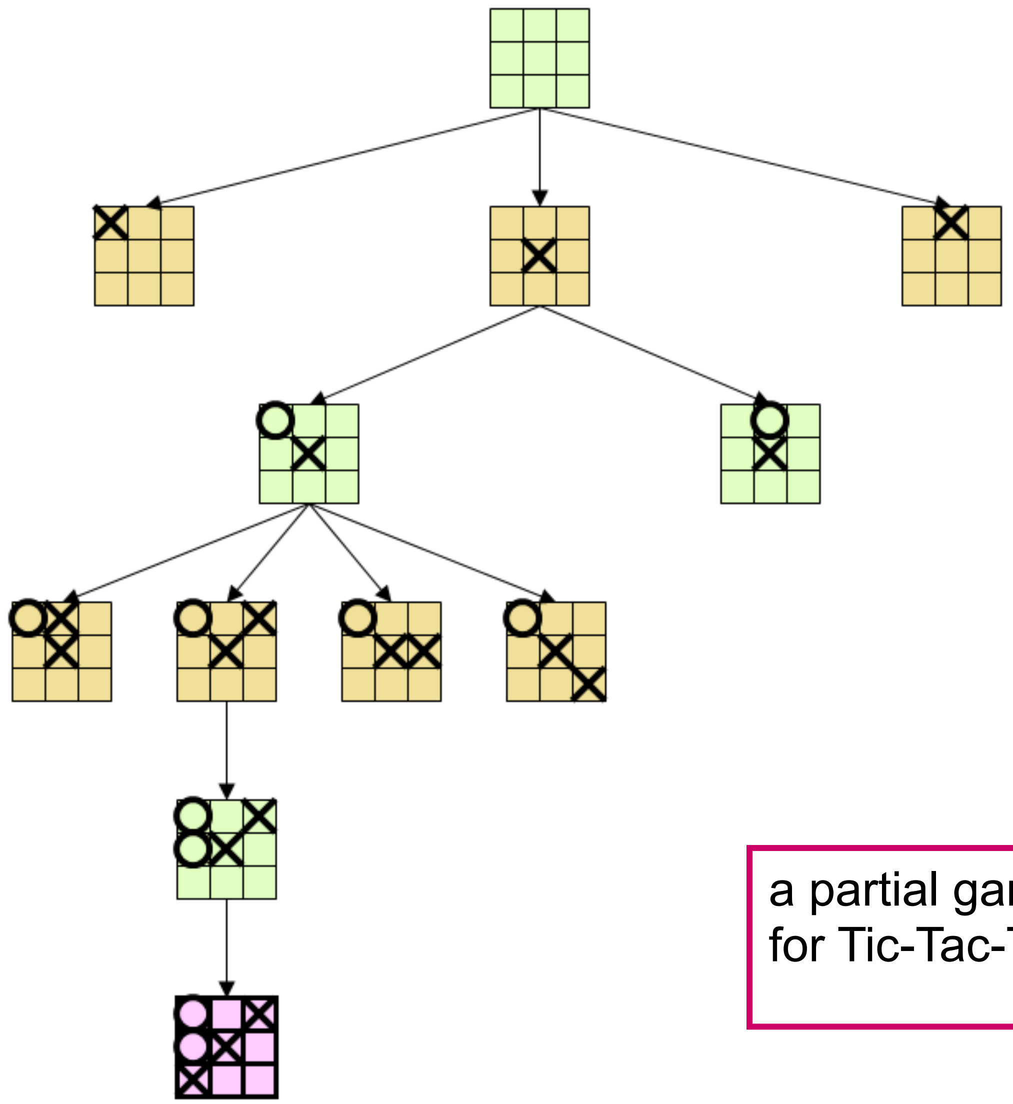
Heuristics (Evaluation function)

- Typical evaluation function for game: weighted linear function
  - $h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_d f_d(s)$
  - *weights* · *features* [dot product]
- For example, in chess
  - $W = \{ 1, 3, 3, 5, 8 \}$
  - $F = \{$ # pawns advantage, # bishops advantage, # knights advantage, # rooks advantage, # queens advantage $\}$
  - Is this what Deep Blue used?
  - What are some problems with this?

Heuristics (Evaluation function)

- Typical evaluation function for game: weighted linear function
  - $h(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_d f_d(s)$
  - *weights* · *features* [dot product]
- For example, in chess
  - $W = \{ 1, 3, 3, 5, 8 \}$
  - $F = \{$ # pawns advantage, # bishops advantage, # knights advantage, # rooks advantage, # queens advantage $\}$
  - Is this what Deep Blue used?
  - What are some problems with this?
- More complex evaluation functions may involve <u>learning</u>
  - Adjusting weights based on outcomes
  - Perhaps non-linear functions
  - How to choose the *features*?

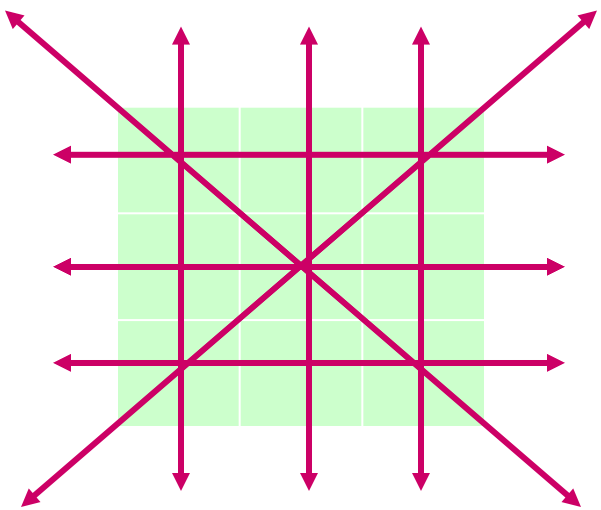
Tic-Tac-Toe revisited



a partial game tree
for Tic-Tac-Toe

Evaluation function for Tic-Tac-Toe

- A simple evaluation function for Tic-Tac-Toe
  - count the number of rows where **X** can win
  - subtract the number of rows where **O** can win
- Value of evaluation function at start of game is zero
  - on an empty game board there are 8 possible winning rows for both **X** and **O**

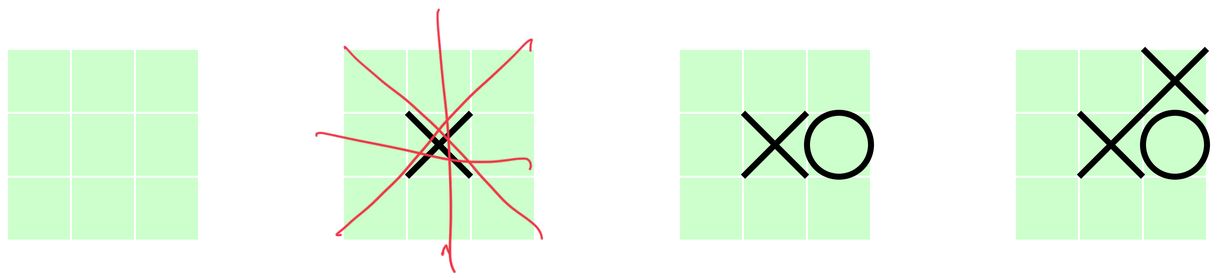


8-8 = 0

Evaluating Tic-Tac-Toe

```
evalX = (number of rows where X can win)  –
                (number of rows where O can win)
```

- After **X** moves in center, score for **X** is +4
- After **O** moves, score for **X** is +2
- After **X**'s next move, score for **X** is +4



8-8 = 0   8-4 = 4   6-4 = 2   6-2 = 4

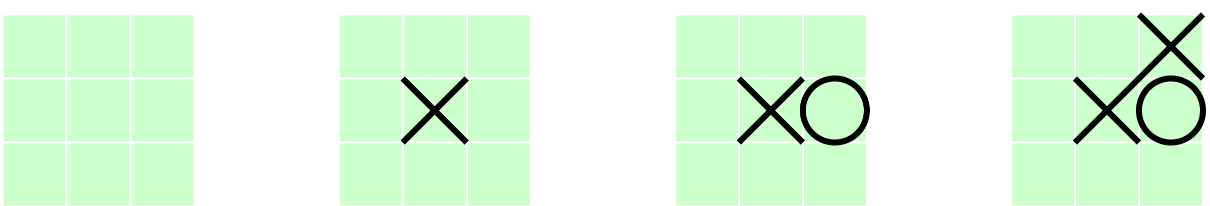
O

# of ways X can win

Evaluating Tic-Tac-Toe

```
evalO = (number of rows where O can win)  –
              (number of rows where X can win)
```

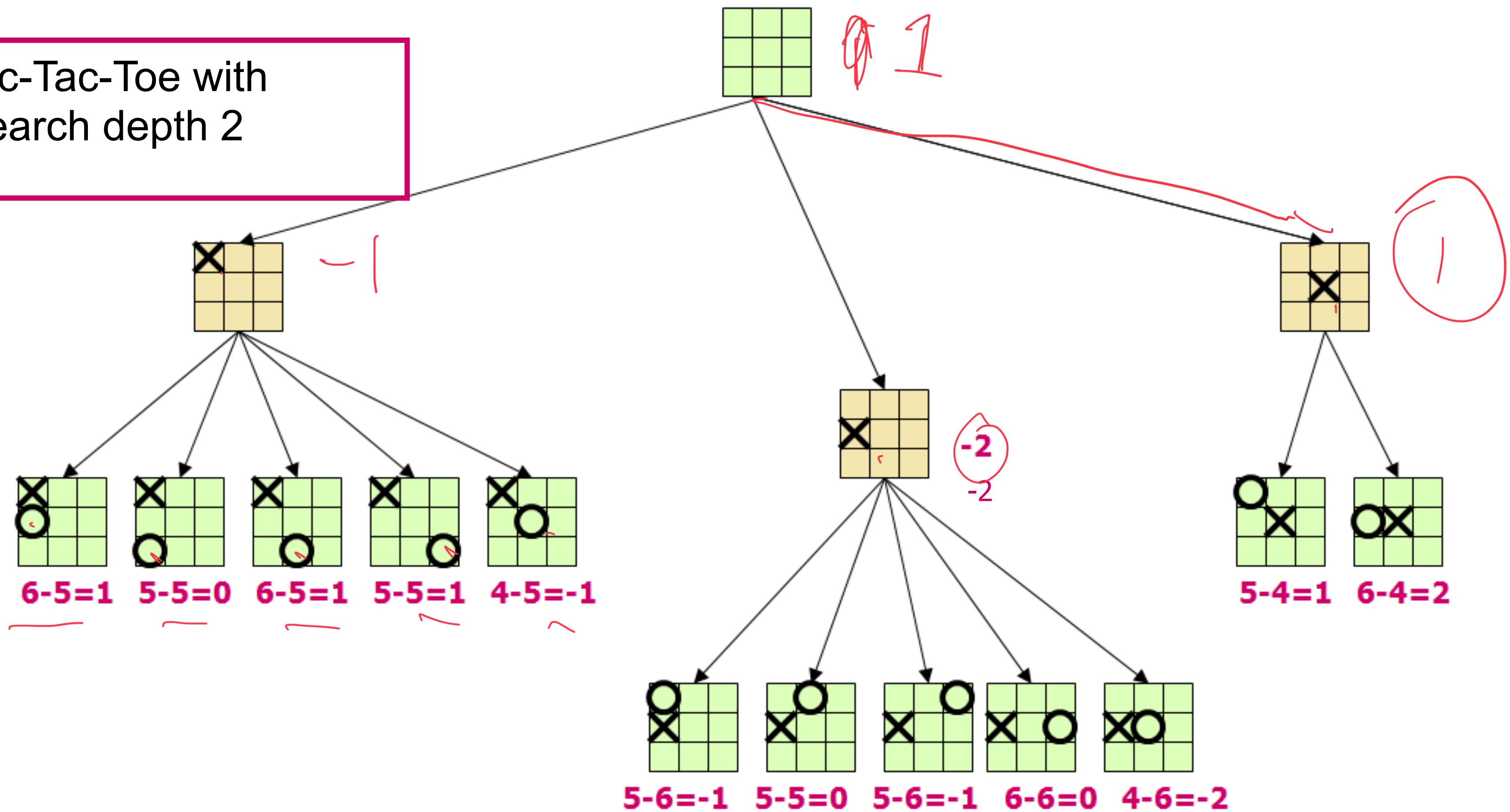
- After **X** moves in center, score for **O** is -4
- After **O** moves, score for **O** is +2
- After **X**'s next move, score for **O** is -4



8-8 = 0   4-8 = -4   4-6 = -2   2-6 = -4

Search depth cutoff

Tic-Tac-Toe with search depth 2



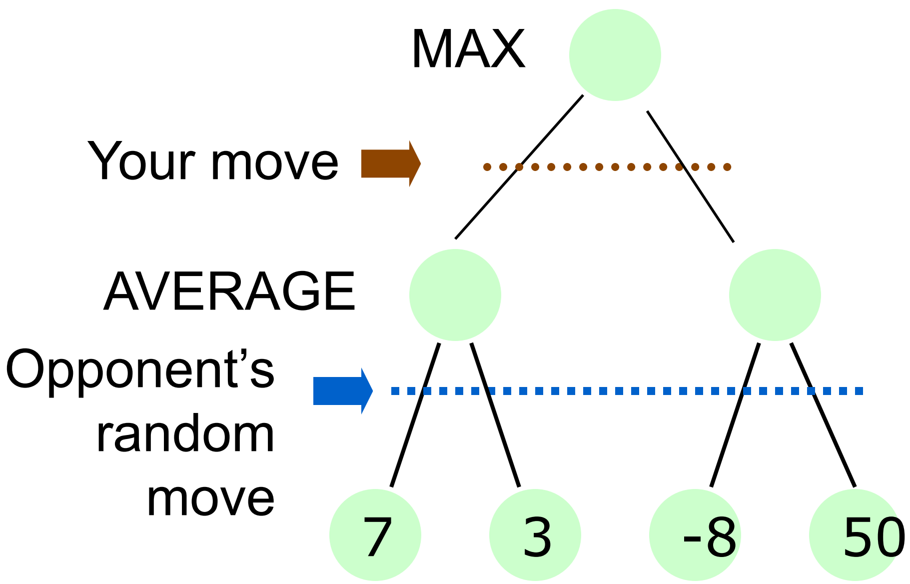
Evaluations shown for X

Expectimax: Playing against a benign opponent

- Sometimes your opponents are not clever.
  - They behave randomly.
  - You can take advantage of that by modeling your opponent.

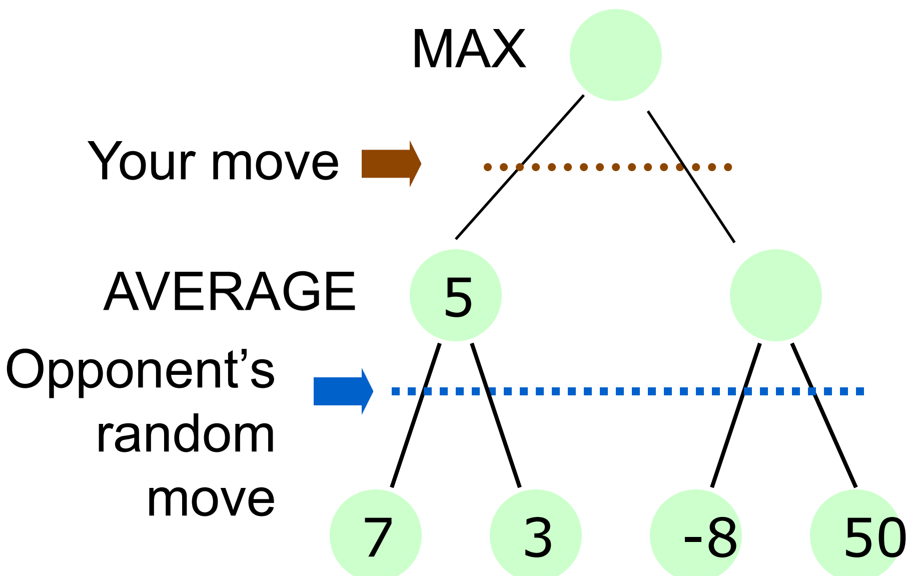
- Example of game of chance:
  - Slot machines
  - Tetris

Expectimax example



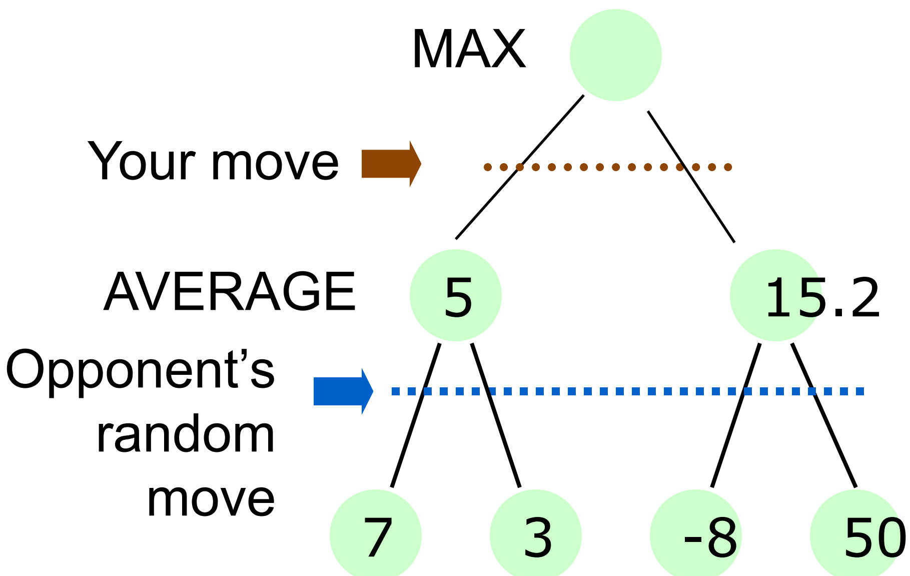
- Your opponent behave randomly with a given probability distribution,
- If you move left, your opponent will select actions with probability [0.5,0.5]
- If you move right, your opponent will select actions with [0.6,0.4]

Expectimax example



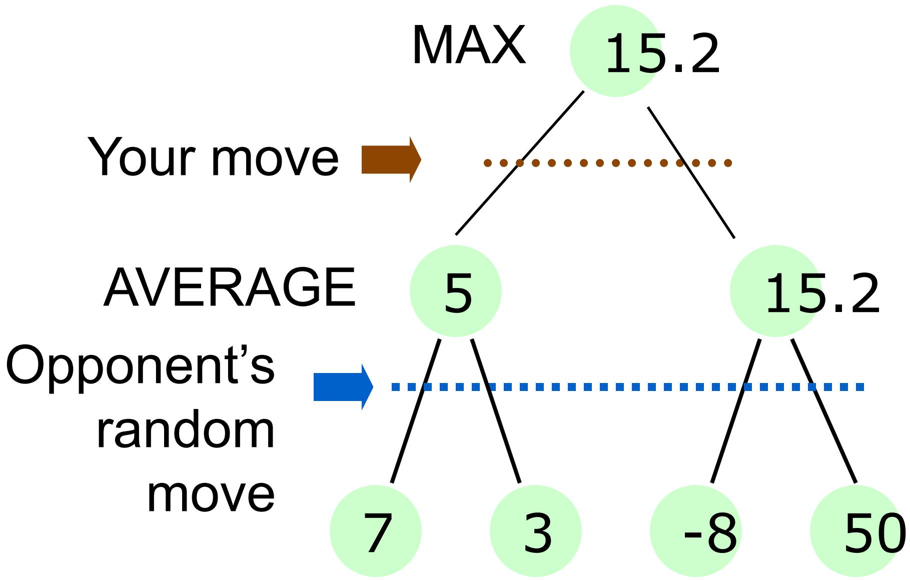
- Your opponent behave randomly with a given probability distribution,
- If you move left, your opponent will select actions with probability [0.5,0.5]
- If you move right, your opponent will select actions with [0.6,0.4]

Expectimax example



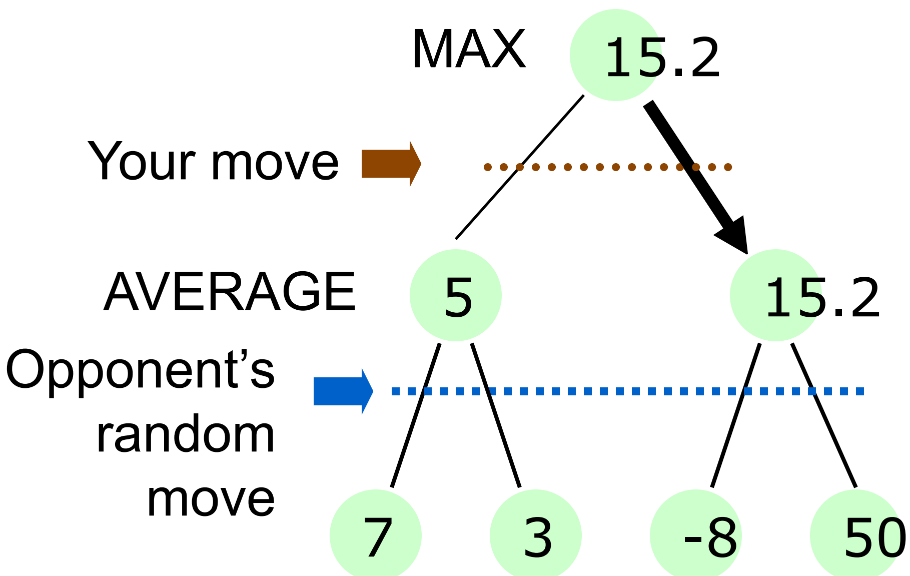
- Your opponent behave randomly with a given probability distribution,
- If you move left, your opponent will select actions with probability [0.5,0.5]
- If you move right, your opponent will select actions with [0.6,0.4]

Expectimax example



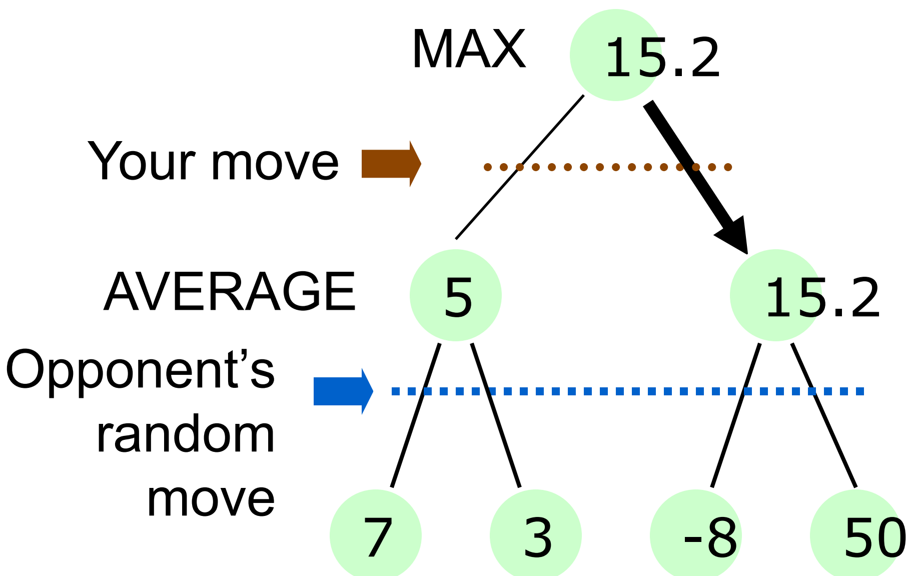
- Your opponent behave randomly with a given probability distribution,
- If you move left, your opponent will select actions with probability [0.5,0.5]
- If you move right, your opponent will select actions with [0.6,0.4]

Expectimax example



- Your opponent behave randomly with a given probability distribution,
- If you move left, your opponent will select actions with probability [0.5,0.5]
- If you move right, your opponent will select actions with [0.6,0.4]

Expectimax example



- Your opponent behave randomly with a given probability distribution,
- If you move left, your opponent will select actions with probability [0.5,0.5]
- If you move right, your opponent will select actions with [0.6,0.4]

Note: pruning becomes tricky in expectimax… think about why.

Summary of game playing

- Minimax search
- Game tree
- Alpha-beta pruning
- Early stop with an evaluation function
- Expectimax

More reading / resources about game playing

- Required reading: AIMA 5.1-5.3

- Stochastic game / Expectiminimax: AIMA 5.5
  - Backgammon. TD-Gammon
  - Blackjack, Poker

- Famous game AI: Read AIMA Ch. 5.7 (or in the "Historical notes" of the AIMA 4th Edition)
  - Deep blue
  - TD Gammon

- AlphaGo: https://www.nature.com/articles/nature16961