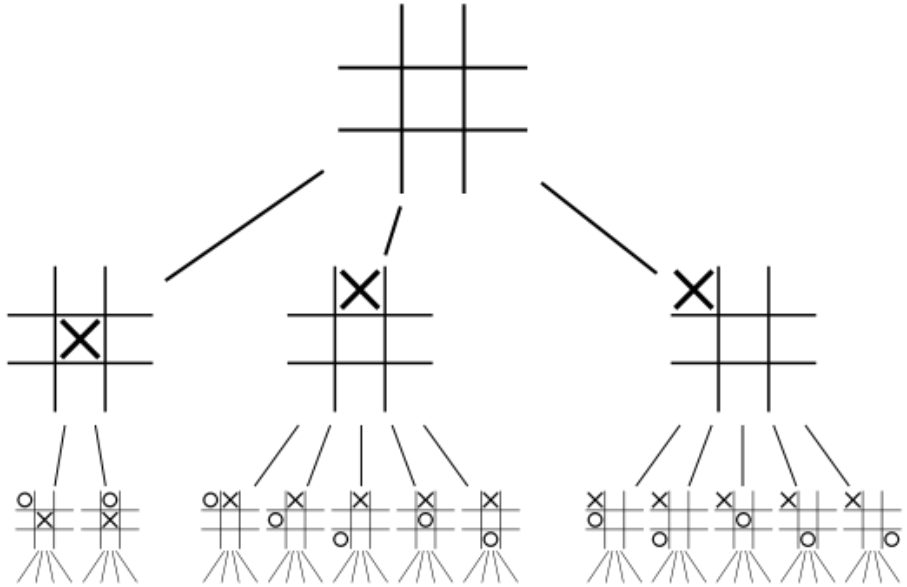
Lecture 2

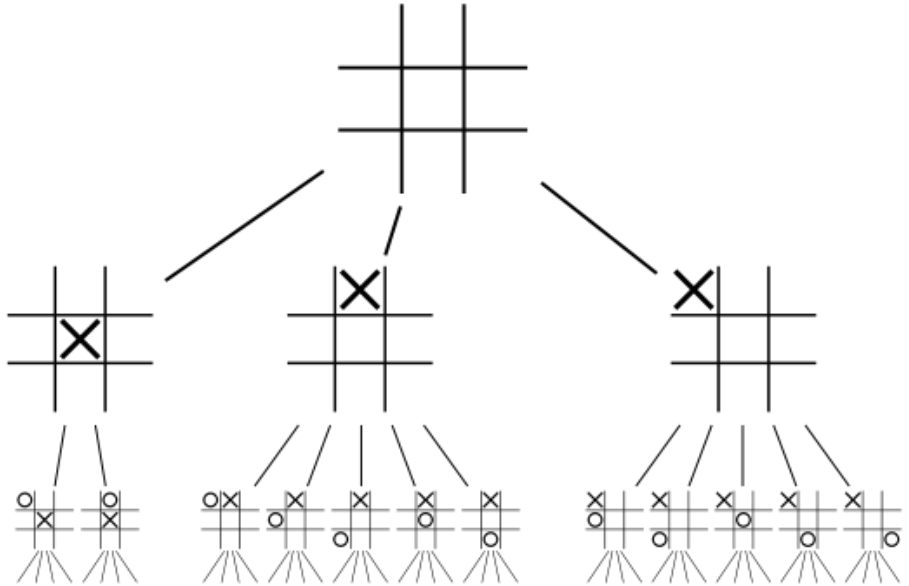
Constructing A State-Tree For Adversarial Search



Source: Wikipedia Commons

Constructing A State-Tree For Adversarial Search

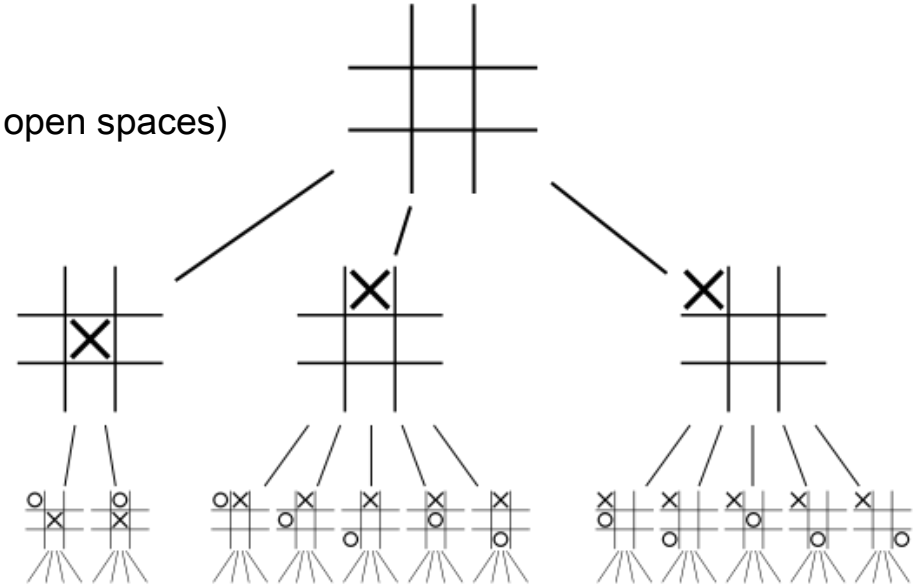
How would A* navigate this?



Source: Wikipedia Commons

Constructing A State-Tree For Adversarial Search

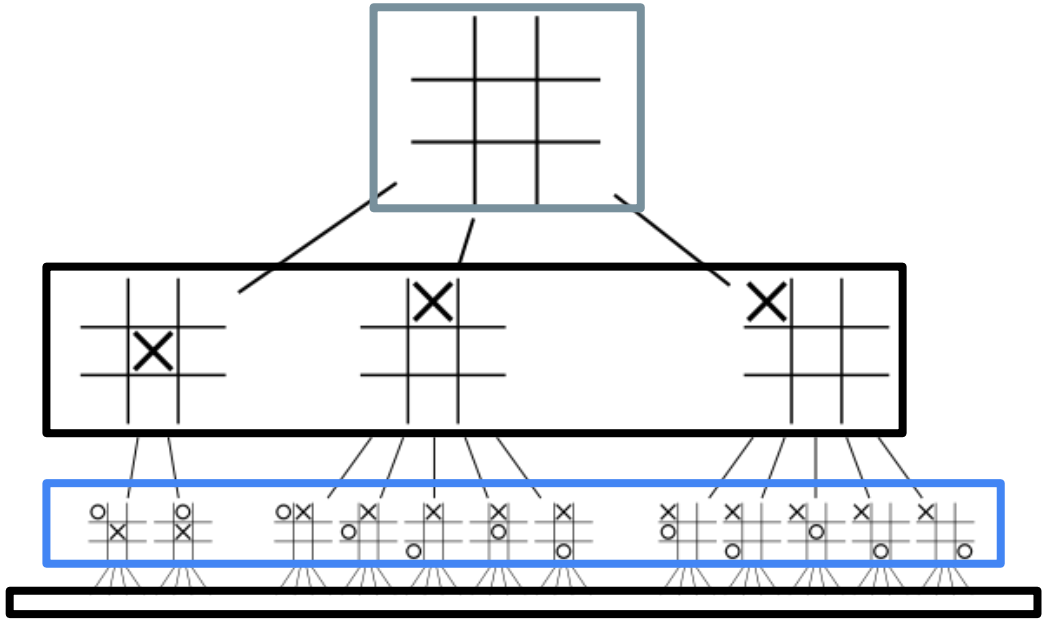
How would A* navigate this?
h(n) = min(3 - largest sequence of x's, # open spaces)



Source: Wikipedia Commons

Constructing A State-Tree For Adversarial Search

Opponent (orange) won't let you (blue) do that



Source: Wikipedia Commons

Lecture 3

Adversarial Search

ECS 170 Summer Session II
University of California, Davis
Gabriel Simmons
(adapted from slides by Michael Livanos)


the rematch
GARRY
KASPAROV
DEEP
BLUE


1:57
1:56


IBM
RS6000 SP

Setting

Limitations of Search

Search frames intelligence as navigation through state space

Search assumes that the agent is in full control

If the world allows a transition, then the agent can make that transition

Limitation of Search

Search frames intelligence as navigation through state space

Search assumes that the agent is in full control

- If the world allows a transition, then the agent can make that transition

Often, multiple agents work against each other **adversarially**

- Games such as chess, checkers, connect 4, Go

Adversarial Search: The Big Idea

Two players take turns manipulating the game state

Each player tries to direct the game tree to their goal state

Adversarial Search Setting

Assumes we have a game with the following conditions:

Two agents (players)

Turn-taking

***Rational* Agents**	Both players take the best move with the information given to them
Deterministic moves	Taking an action at a state always returns the same new state
Perfect information	No “hidden” parts of the state. Both players know everything that is happening
Zero-sum	What’s good for you is equally bad for your opponent and vise versa

Adversarial Search: Naive Approach

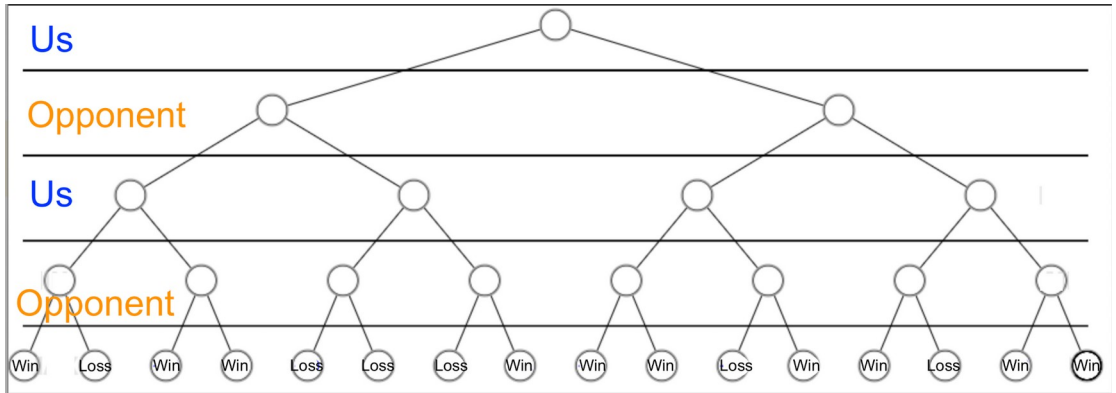
Generate the whole game tree

- If the game is over at a particular state, the node is ***terminal***

Assume you and your adversary are <u>rational</u>

- Assume you will move the game to a win if you have the opportunity
- Assume opponent will move the game to a loss if they have the opportunity

Work backwards from terminal states to choose your move



Adversarial Search: Naive Approach

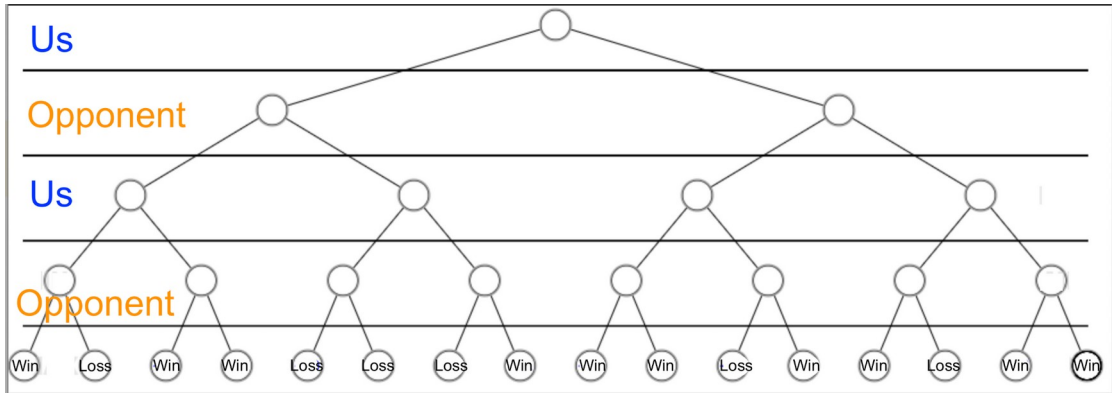
Generate the whole game tree

- If the game is over at a particular state, the node is ***terminal***

Assume you and your adversary are <u>rational</u>

- Assume you will move the game to a win if you have the opportunity
- Assume opponent will move the game to a loss if they have the opportunity

Work backwards from terminal states to choose your move



Adversarial Search: Naive Approach

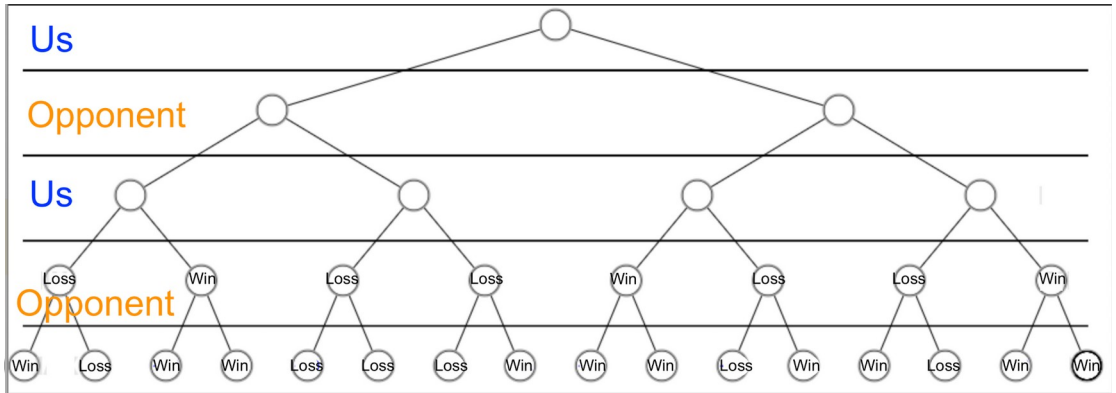
Generate the whole game tree

- If the game is over at a particular state, the node is ***terminal***

Assume you and your adversary are <u>rational</u>

- Assume you will move the game to a win if you have the opportunity
- Assume opponent will move the game to a loss if they have the opportunity

Work backwards from terminal states to choose your move



Adversarial Search: Naive Approach

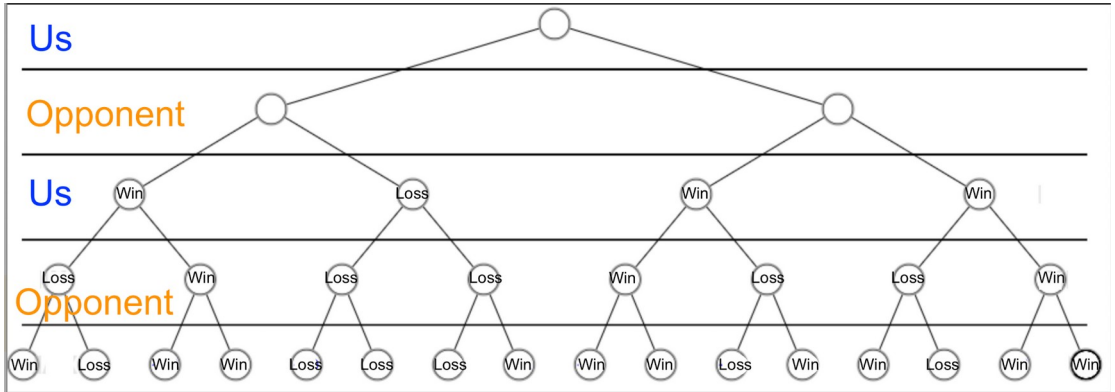
Generate the whole game tree

- If the game is over at a particular state, the node is ***terminal***

Assume you and your adversary are <u>rational</u>

- Assume you will move the game to a win if you have the opportunity
- Assume opponent will move the game to a loss if they have the opportunity

Work backwards from terminal states to choose your move



Adversarial Search: Naive Approach

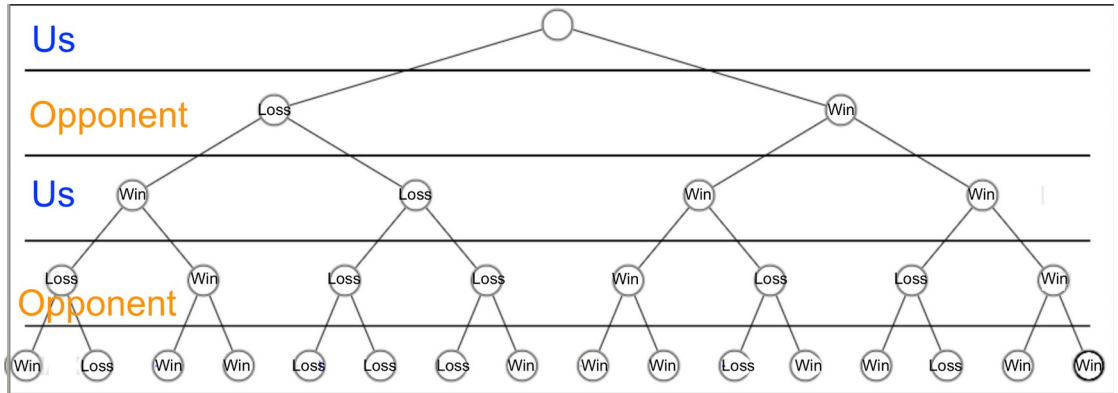
Generate the whole game tree

- If the game is over at a particular state, the node is ***terminal***

Assume you and your adversary are <u>rational</u>

- Assume you will move the game to a win if you have the opportunity
- Assume opponent will move the game to a loss if they have the opportunity

Work backwards from terminal states to choose your move



Adversarial Search: Naive Approach

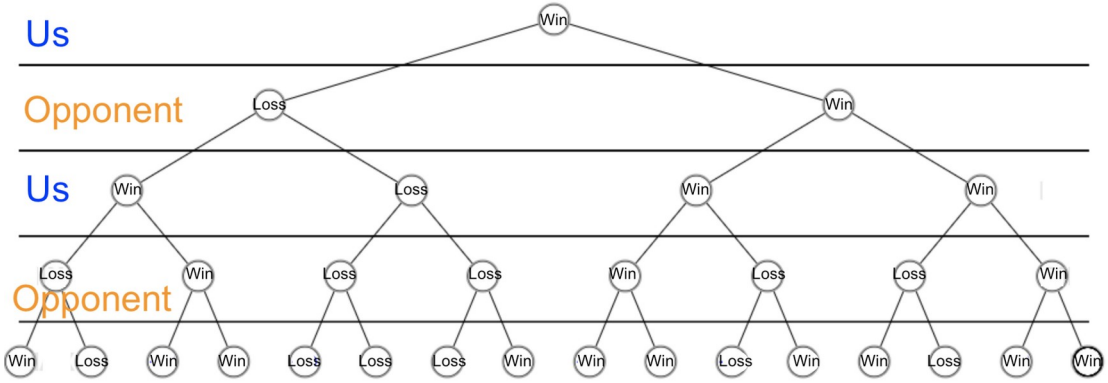
Generate the whole game tree

- If the game is over at a particular state, the node is ***terminal***

Assume you and your adversary are <u>rational</u>

- Assume you will move the game to a win if you have the opportunity
- Assume opponent will move the game to a loss if they have the opportunity

Work backwards from terminal states to choose your move



Adversarial Search: Naive Approach

Issues?

Adversarial Search: Naive Approach

Issue: Assumes we can generate the whole tree

Tree grows b^d

Tic-Tac-Toe	b = 4, d = 9	~ 260,000
Connect 4	b = 4, d = 42	~ $2 * 10^{25}$
Chess	b = 35, d = 40	35^{40} ~ $5.7 * 10^{61}$
Go	b = 19x19 = 361, d ~180	361^{180} ~ 10^{460} (Number of legal gamestates estimated 10^{360}*)

Branching factor source

*J. M. Robson (1983). "The complexity of Go". *Information Processing; Proceedings of IFIP Congress*. pp. 413–417.

Adversarial Search: Minimax

Solution:

Build a tree as deep as we can afford (less than full depth)

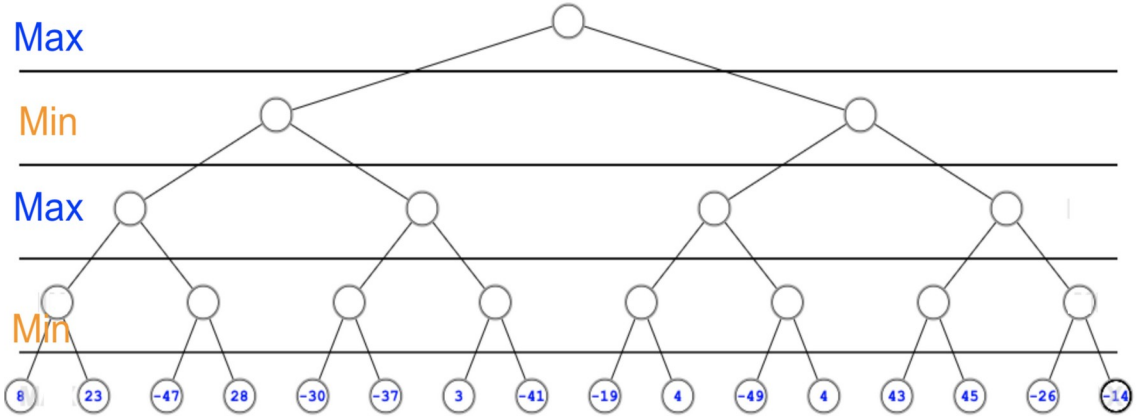
then…

evaluate how good the terminal states look

Adversarial Search: Minimax

Modifications to Naive Adversarial Search

1. Replace win/loss with an evaluation
2. Replace binary propagation with min (opponent) and max (us).

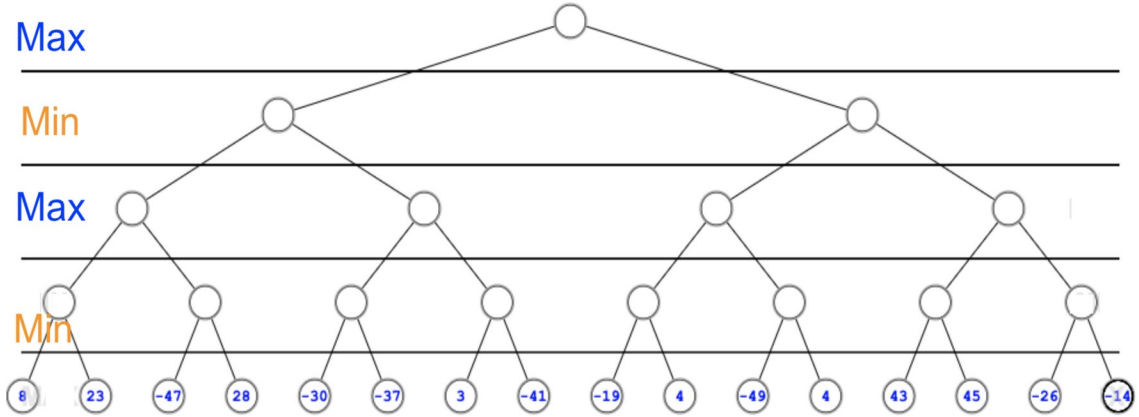


Adversarial Search: Minimax

Modifications to Naive Adversarial Search

1. Replace win/loss with an evaluation
2. Replace binary propagation with min (opponent) and max (us).

example



Adversarial Search: Minimax

Modifications to Naive Adversarial Search

1. Replace win/loss with an evaluation
2. Replace binary propagation with min (opponent) and max (us).

Discussion: How intelligent is Minimax?

Adversarial Search: Minimax

Modifications to Naive Adversarial Search

1. Replace win/loss with an **evaluation**
2. Replace binary propagation with min (opponent) and max (us).

Note: We can adjust performance by modifying the depth

Designing Evaluation Functions

What is an evaluation function?

Estimate of how “good” a state is for us

What is an evaluation function?

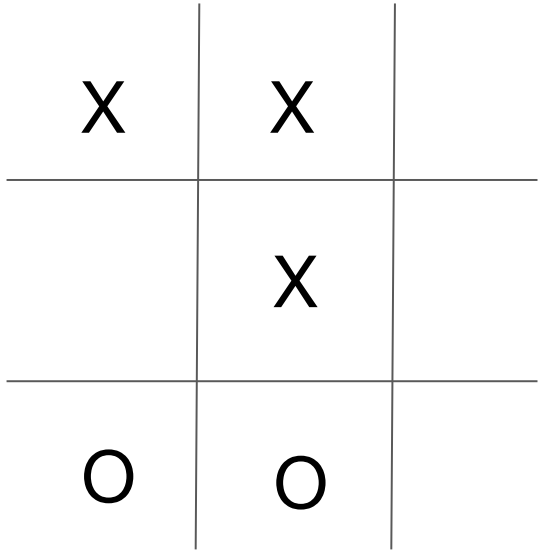
Estimate of how “good” a state is for us

good for us == bad for opponent (*zero sum*)

Wait, is this a heuristic like in A*?

- Not really - it is a guess about the state, but it does not need to be admissible
- The good - an evaluation function can be anything, no proof required
- The bad - no guarantee of optimality

Discussion: How Can We Design An Evaluation Function For Tic-Tac-Toe

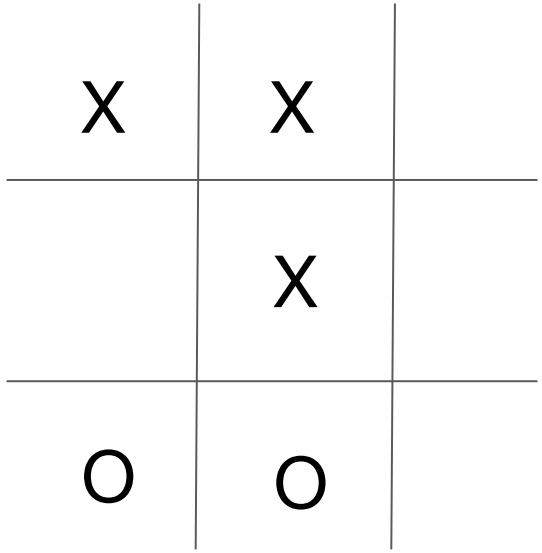


Discussion: How Can We Design An Evaluation Function For Tic-Tac-Toe

Two relevant factors:

- Our pieces
- Opponent pieces

eval(S) = ways to win - ways to lose



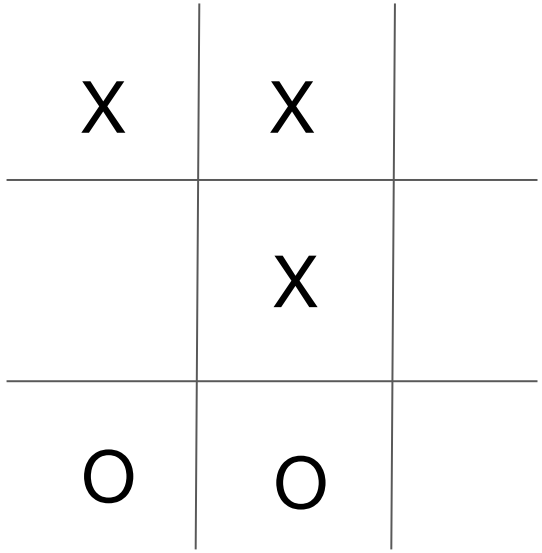
Discussion: How Can We Design An Evaluation Function For Tic-Tac-Toe

Two relevant factors:

- Our pieces
- Opponent pieces

eval(S) = ways to win - ways to lose

eval(S) = 1 if win, -1 if loss, 0 if tied



How do we create evaluation functions?

Two approaches:

quality

Expensive Analysis

+ High fidelity

- Can’t look down as far

quantity

Simple Analysis

+ Cheap to evaluate, can cover a large part of the tree
- Low fidelity

Deep Blue: A mix of both

What Do We Need For Deep Blue?

Minimax

???

???

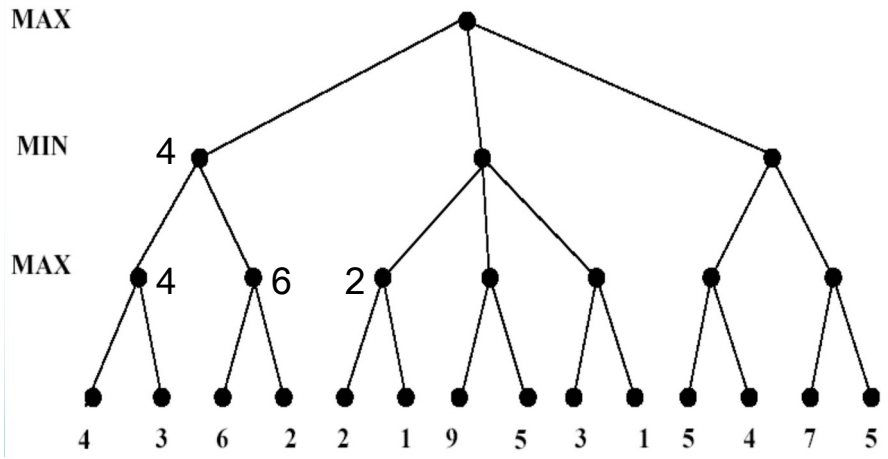
Alpha-Beta Pruning

Limitations of Minimax

Tree grows as $O(b^d)$

We are evaluating every node. Do we have to?

b: branching factor
d: depth

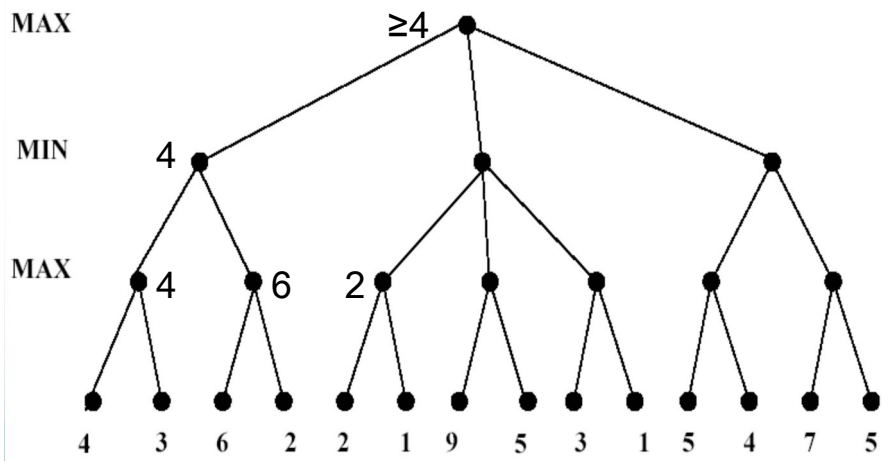


Limitations of Minimax

Tree grows as $O(b^d)$

We are evaluating every node. Do we have to?

b: branching factor
d: depth

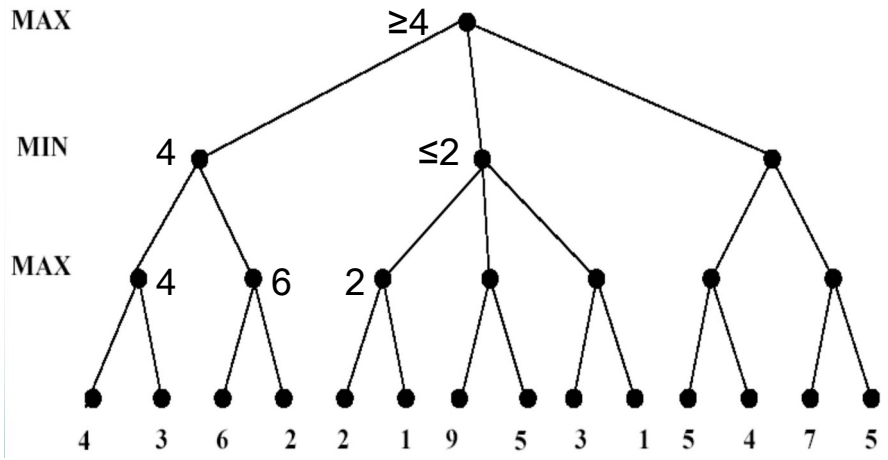


Limitations of Minimax

Tree grows as $O(b^d)$

We are evaluating every node. Do we have to?

b: branching factor
d: depth



Limitations of Minimax

Tree grows as $O(b^d)$

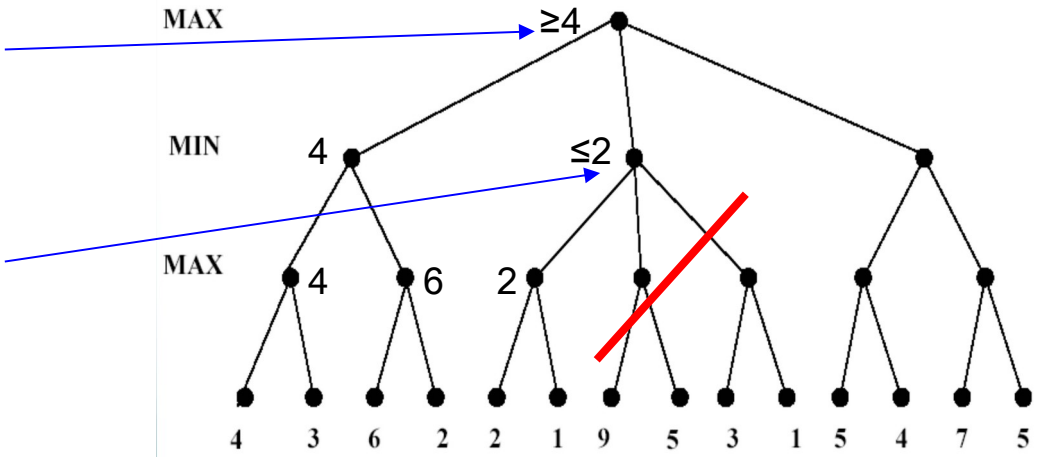
We are evaluating every node. Do we have to?

b: branching factor
d: depth

We can get a value of *at least 4* for the game…

If we go this way, *our adversary won't let us get more than 2…*

So we don't go that way - no need to keep evaluating this branch



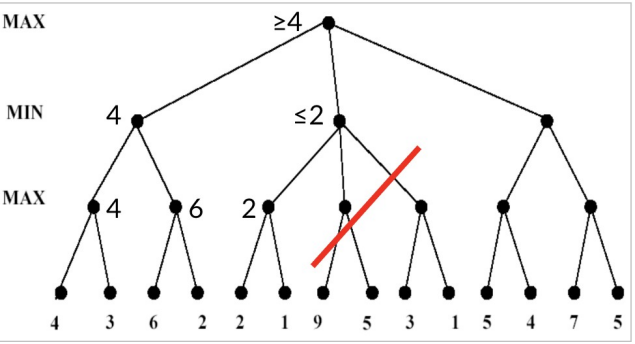
Pruning Unnecessary Evaluations

Recall:

Max: Maximizes the minimum guaranteed reward

Min: Minimizes the maximum guaranteed reward

We need to keep track of…



Our best case value so far
(higher is better for us)

Our adversary's best case value so far
(lower is better for them)

Minimax Algorithm

```
def Max(state, depth):
        if GameOver(state) || depth == max_depth:
                return eval(state)

    value = -infinity

        for child in state.children():
                value = max(value, Min(child, depth + 1))

        return value

def Min(state, depth):
        if GameOver(state) || depth == max_depth:
                return eval(state)

        value = infinity

        for child in state.children():
                value = min(value, Max(child, depth + 1))

        return value
```

Minimax With Alpha-Beta Pruning

```
def Max(state, depth, alpha, beta):
        if GameOver(state) || depth == max_depth:
                return eval(state)

    value = -infinity

        for child in state.children():
                value = max(value, Min(
                                child,
                                depth + 1,
                                alpha,
                                beta
                        )
                )
                alpha = max(alpha, value)
        if value ≥ beta:
                        break

        return value
```

```
def Min(state, depth, alpha, beta):
        if GameOver(state) || depth == max_depth:
                return eval(state)

        value = infinity

        for child in state.children():
                value = min(value, Max(
                                child,
                                depth + 1,
                                alpha,
                                Beta
                        )
                )
                beta = min(beta, value)
                if value ≤ alpha:
                        break

        return value
```

Minimax With Alpha-Beta Pruning

```
def Max(state, depth, alpha, beta):
        if GameOver(state) || depth == max_depth:
                return eval(state)

    value = -infinity

        for child in state.children():
                value = max(value, Min(
                        child,
                        depth + 1,
                        alpha,
                        beta
                    )
            )
                alpha = max(alpha, value)
        if value ≥ beta:
                break

        return value
```

The value of this state is the maximum of the successor values

update our overall best-case value

If value for this state is greater than adversary's best-case value, stop evaluating

Analysis: Performance Gain

For equal depth, Alpha-Beta Pruning will perform the *same gameplay as minimax, with less computational effort*

Recall: Minimax is $O(b^d)$ complexity

What is the complexity of alpha-beta pruning?

- How much pruning can occur?
- How much pruning tends to occur?
- Is any pruning guaranteed?

How Much Pruning Can Occur?

From Max's perspective (us):

Max for state S is the maximum of the Min of the successor states

$$\mathrm{MAX}(S) = \max(\ \mathrm{MIN}(S'_0),\ \mathrm{MIN}(S'_1),\ \mathrm{MIN}(S'_2),\ \ldots\ )$$

We start by evaluating the first child to compute $\mathbf{V_0}$

$$\mathrm{MAX}(S) = \max(\ \mathbf{V_0},\ \mathrm{MIN}(S'_1),\ \mathrm{MIN}(S'_2),\ \ldots\ )$$

If the upper bound of another child is less than or equal to V_0, we prune

Most pruning occurs for Max when successor states are evaluated in ***descending order of value***

How Much Pruning Can Occur?

From Min's perspective (our adversary):

Min for state S is the minimum of the Max of the successor states

$$\text{Min}(S) = \min(\ \text{Max}(S'_0),\ \text{Max}(S'_1),\ \text{Max}(S'_2),\ \ldots\ )$$

We start by evaluating the first child to compute $\mathbf{V_0}$

$$\text{Min}(S) = \min(\ \mathbf{V_0},\ \text{Max}(S'_1),\ \text{Max}(S'_2),\ \ldots\ )$$

If the lower bound of another child is greater than or equal to V_0, we prune

Most pruning occurs for Min when successor states are evaluated in ***ascending order of value***

How Much Pruning Can Occur?

Most pruning occurs for **Max** when successor states are evaluated in **descending order of value**

Most pruning occurs for Min when successor states are evaluated in **ascending order of value**

Optimal pruning occurs when Min and Max evaluate what is best for them first

How Much Pruning Can Occur?

Minimax complexity:

$O(b^d) = O(b*b*b\ldots)$

If we have *optimal pruning*, we must evaluate one child to completion, and we get to prune the rest

Alpha beta complexity with optimal pruning (best case):

$O(b*1*b*1*...) = O(b^{d/2})$

How Much Pruning Can Occur?

We can’t guarantee optimal pruning – If we knew the best move, we wouldn’t be searching for it

Is Pruning Guaranteed To Occur?

From Max's perspective (us):

Max for state S is the maximum of the Min of the successor states

$$\mathrm{MAX}(S) = \max(\ \mathrm{MIN}(S'_0),\ \mathrm{MIN}(S'_1),\ \mathrm{MIN}(S'_2),\ \ldots\ )$$

We start by evaluating the first child to compute $\mathbf{V_0}$

$$\mathrm{MAX}(S) = \max(\ \mathbf{V_0},\ \mathrm{MIN}(S'_1),\ \mathrm{MIN}(S'_2),\ \ldots\ )$$

If the upper bound of another child is greater than or equal to V_0, we don't prune

No pruning occurs for Max when successor states are evaluated in ***ascending order of value***

Is Pruning Guaranteed To Occur?

No pruning occurs for Max when successor states are evaluated in **ascending order of value**

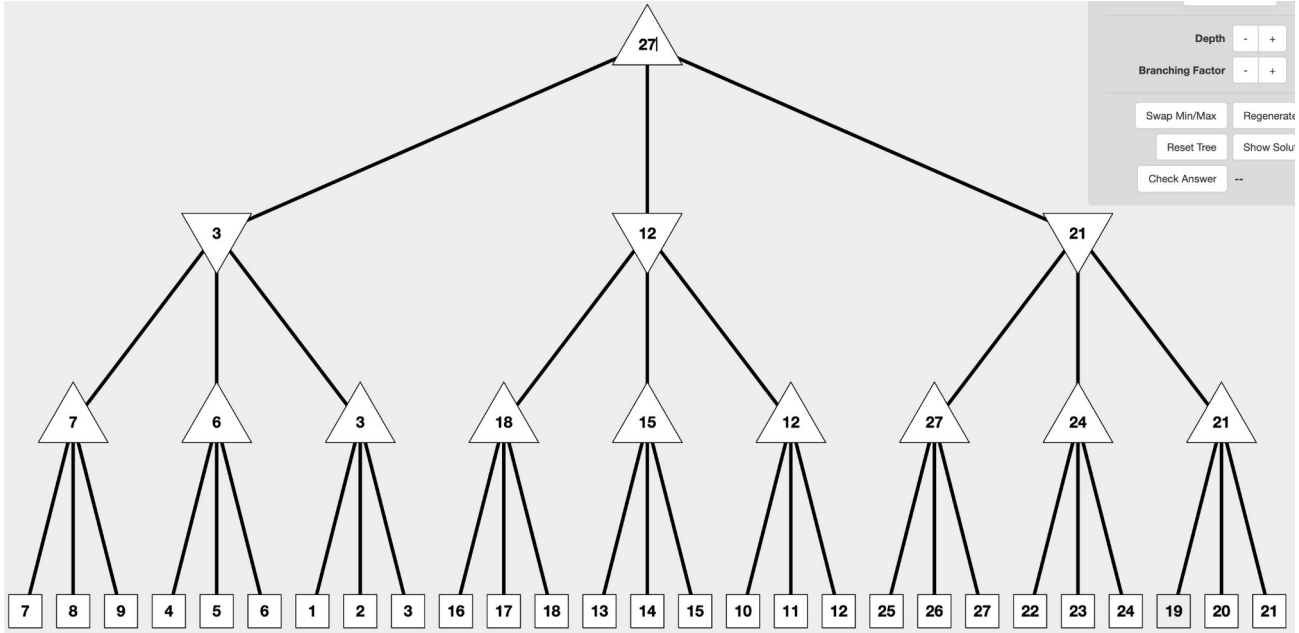
No pruning occurs for Min when successor states are evaluated in **descending order of value**

If Max and Min evaluate what's worst for them first, no pruning occurs

Pruning is not guaranteed to occur

Alpha beta complexity with no pruning (worst case):

$O(b^d) = O(b*b*b\ldots)$


27
3
12
21
7
6
3
18
15
12
27
24
21
7 8 9 4 5 6 1 2 3 16 17 18 13 14 15 10 11 12 25 26 27 22 23 24 19 20 21
Depth
Branching Factor
Swap Min/Max
Reset Tree
Check Answer
--

Analysis: Performance Gain

Minimax is $O(b^d)$ complexity

What is the complexity alpha-beta pruning?

- How much pruning can occur? Half the tree
- How much pruning tends to occur?
- Is any pruning guaranteed? No

How much pruning tends to occur? Between none and half the tree.

Alpha-Beta Complexity (typical case): somewhere between $O(b^d)$ and $O(b^{d/2})$, inclusive

How much pruning tends to occur?

It's complicated

- Russell & Norvig says $O(b^{3d/4})$
- Some researchers disagree
- General agreement is that it is closer to $O(b^{d/2})$, $O((b/\log(b))^d)$ is common

Perhaps a better question is…

how can we get closer to perfect pruning?

How Can We Push Alpha-Beta to Optimal Pruning?

- Can we rearrange the tree after we see the children?
  - No, that implies that we generate the entire tree which is $O(b^d)$ complexity

How Can We Push Alpha-Beta to Optimal Pruning?

- Can we rearrange the tree after we see the children?
  - No, that implies that we generate the entire tree which is $O(b^d)$ complexity
- Use knowledge about the problem to order nodes statically
  - Tic-tac-toe: Middle is best spot, corners are better
  - Chess: Taking a piece using a pawn tends to be good

How Can We Push Alpha-Beta to Optimal Pruning?

- Can we rearrange the tree after we see the children?
  - No, that implies that we generate the entire tree which is $O(b^d)$ complexity
- Use knowledge about the problem to **order nodes statically**
  - Tic-tac-toe: Middle is best spot, corners are better
  - Chess: Taking a piece using a pawn tends to be good
- Use the evaluation function to order nodes dynamically:
  - Issue: If evaluation function is complicated, we might lose our performance benefits
  - Solution: We could use a simple evaluation function for this purpose

What Do We Need For Deep Blue?

…with AlphaBeta pruning

Minimax

???

Optimizing Alpha-Beta Search: Working Under Time Constraints

Iterative Deepening

- Suppose we craft an evaluation function that can look 5 moves ahead 98% of the time and 6 moves ahead 75% of the time

Iterative Deepening

Suppose we craft an evaluation function that can look 5 moves ahead 98% of the time and 6 moves ahead 75% of the time

How should we set depth?

- Play it safe? Look 5 moves ahead and get moves almost always?
- Be adventurous? Go for 6!

How Expensive Is Iterative Deepening?

Cost of alpha-beta pruning: $O(b^{3d/4})$

Cost of alpha-beta pruning down 6 moves:

$e*b^{4.5}$

e: cost of evaluation

Cost of alpha-beta pruning down 5 moves:

$e*b^{3.75}$

How Expensive Is Iterative Deepening?

Branching Factor	Cost at 6	Cost at 5	% Difference
7 (Connect 4)	6352 * e	1477*e	22%
35 (Chess)	8,887,000 * e	617,000 * e	6.9%
361 (Go)	$3.23 * 10^{11} * e$	$3.89 * 10^{9} * e$	1.2%

Note: Difference gets smaller with greater depths

How Expensive Is Iterative Deepening?

Branching Factor	Cost at 6	Cost to deepen all the way to root	% Difference
7 (Connect 4)	6352 * e	1921*e	30%
35 (Chess)	8,887,000 * e	663,000 * e	7.5%
361 (Go)	$3.23 * 10^{11} * e$	$3.94 * 10^{9} * e$	1.22%

Note: Difference gets smaller with greater depths

But I Don’t Want Any Extra Cost!

- This can be another way to order nodes and ensure optimal pruning!
  - Keep track of which nodes were best, and when we redo the tree, order children in this way

But I Don’t Want Any Extra Cost!

- This can be another way to order nodes and ensure optimal pruning!
  - Keep track of which nodes were best, and when we redo the tree, order children in this way
  - Outperforms previous techniques
    - Just as evaluating each child is not as good as looking deeper down the tree, ordering based on iterative deepening is better than a simple evaluation

But I Don’t Want Any Extra Cost!

- This can be another way to order nodes and ensure optimal pruning!
  - Keep track of which nodes were best, and when we redo the tree, order children in this way
  - Outperforms previous techniques
    - Just as evaluating each child is not as good as looking deeper down the tree, ordering based on iterative deepening is better than a simple evaluation
- In practice, this technique tends to recuperate its cost and more

Other Optimization Techniques

- Caching the tree between runs
  - After alpha-beta pruning, two moves take place (one from your agent, one for the opponent)

Other Optimization Techniques

- Caching the tree between runs
  - After alpha-beta pruning, two moves take place (one from your agent, one for the opponent)
  - That means that you have d-2 moves of the tree already made if you store

Other Optimization Techniques

- Caching the tree between runs
  - After alpha-beta pruning, two moves take place (one from your agent, one for the opponent)
  - That means that you have d-2 moves of the tree already made if you store
- Evaluating a tree while opponent is thinking
  - You might not know which branch the opponent will chose, but you can generate more of the tree and prune the paths not used
  - Not efficient, but the alternative is doing nothing and wasting time

What Do We Need For Deep Blue?

…with AlphaBeta pruning

Minimax

& iterative deepening

What Do We Need For Deep Blue?

…with AlphaBeta pruning

Minimax

& iterative deepening

Multiprocessing

Streamlined evaluation functions

Slow and fast evaluation functions

Applications, Limitations, & Variations

Building Deep Blue (1997)

- We now know enough about AI to build Deep-Blue!
- Parallelized minimax using
  - Alpha-beta pruning
  - Iterative deepening
- Evaluation function:
  - Slow evaluation function (complicated)
  - Fast evaluation function (single clock cycle, simple)
  - You can read more about it if you want

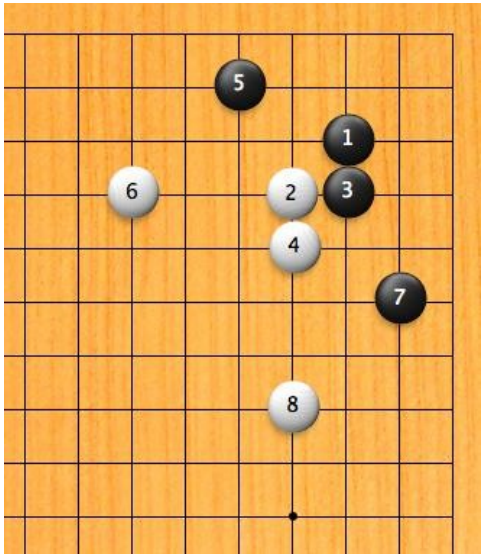
Alpha-Beta Search Today

- Still used by highly ranked chess playing agents such as stockfish
- Can create extremely good agents for almost any game that meets the requirements

Alpha-Beta Search Today

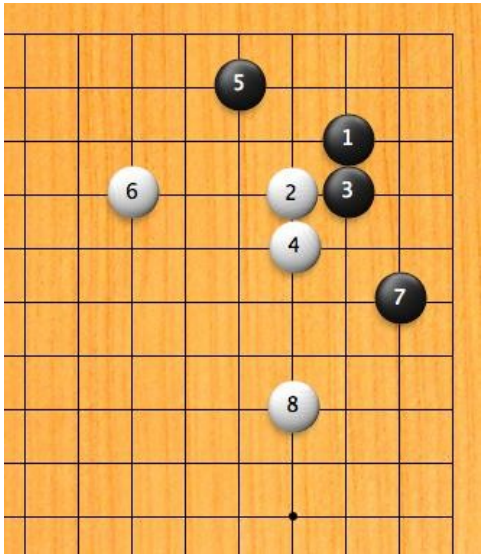
- Still used by highly ranked chess playing agents such as stockfish
- Can create extremely good agents for almost any game that meets the requirements

Alpha-Beta Search Today



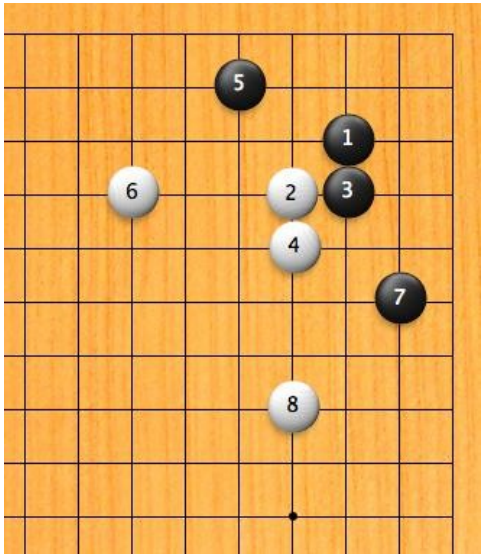
- Still used by highly ranked chess playing agents such as stockfish
- Can create extremely good agents for almost any game that meets the requirements
    - Almost? Recall the discussion on Go. Looking 6 moves ahead cost ~400 Billion evaluations

Alpha-Beta Search Today



- Still used by highly ranked chess playing agents such as stockfish
- Can create extremely good agents for almost any game that meets the requirements
  - Almost? Recall the discussion on Go. Looking 6 moves ahead cost ~400 Billion evaluations
  - Beginner and intermediate players know Joseki, but these frequently require to look 8 moves ahead
  - Costs ~$2.2*10^{15}$ evaluations

Alpha-Beta Search Today



- Still used by highly ranked chess playing agents such as stockfish
- Can create extremely good agents for almost any game that meets the requirements
  - Almost? Recall the discussion on Go. Looking 6 moves ahead cost ~400 Billion evaluations
  - Beginner and intermediate players know Joseki, but these frequently require to look 8 moves ahead
  - Costs ~$2.2*10^{15}$ evaluations
- But this isn't how Go players play. Not every space is relevant to every move

Limitations of Alpha-Beta Search

- Too costly for huge branching factor problems
  - Can’t play games like Go or games with continuous actions spaces well

Limitations of Alpha-Beta Search

- Too costly for huge branching factor problems
  - Can’t play games like Go or games with continuous actions spaces well
- Recall requirements for minimax adversarial search:

Limitations of Alpha-Beta Search

- Too costly for huge branching factor problems
  - Can’t play games like Go or games with continuous actions spaces well
- Recall requirements for minimax adversarial search:
  - Two player
  - Turn taking
  - Rational agents
  - Deterministic
  - Perfect information
  - Zero-Sum

Limitations of Alpha-Beta Search

- Too costly for huge branching factor problems
  - Can’t play games like Go or games with continuous actions spaces well
- Recall requirements for minimax adversarial search:
  - Two player
  - Turn taking
  - Rational agents
  - Deterministic
  - Perfect information
  - Zero-Sum

Dealing With 2-Player Limit: Max^n

- What about games with >2 players?

Dealing With 2-Player Limit: Max^n

- What about games with >2 players?
  - Consider us Max and both players Min?

Dealing With 2-Player Limit: Max^n

- What about games with >2 players?
  - Consider us Max and both players Min?
    - Violates rational agent assumption

Dealing With 2-Player Limit: Max^n

- What about games with >2 players?
  - Consider us Max and both players Min?
    - Violates rational agent assumption
- Recall: Min wants to minimize maximum guaranteed reward
- Recall: Zero-sum principle - What is good for us is equally bad for opponent

Dealing With 2-Player Limit: Max^n

- What about games with >2 players?
  - Consider us Max and both players Min?
    - Violates rational agent assumption
- Recall: Min wants to minimize maximum guaranteed reward
- Recall: Zero-sum principle - What is good for us is equally bad for opponent
- Min is just Max for negative evaluation function

Dealing With 2-Player Limit: Max^n

- Max^n:
  - Each player is maximizing an evaluation
  - V_{p1} = eval(P1) - eval(P2) / (n-1) - eval(P3) / (n-1)
  - eval'(player) is the evaluation of that player from their prospective
  - Eg in connect4 we have a sequence-based evaluation function, find our sequences, weight them and subtract that of opponent

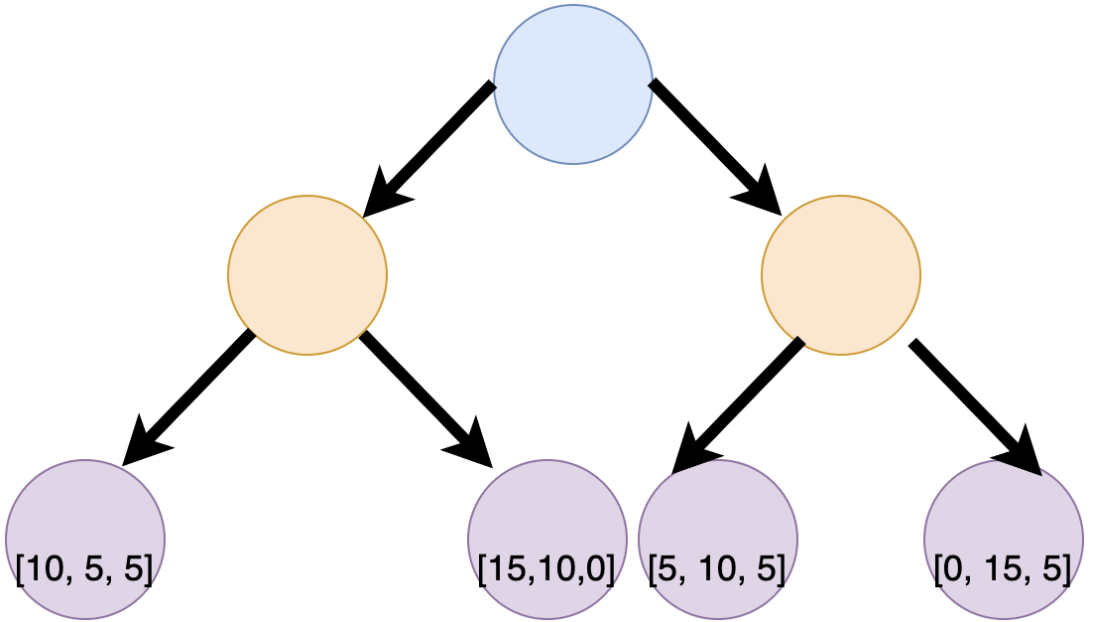
Dealing With 2-Player Limit: Max^n

- Max^n:
  - Each player is maximizing an evaluation
  - V_{p1} = eval(P1) - (eval(P2) + eval(P3) + …) / (n-1)
  - eval'(player) is the evaluation of that player from their prospective
  - Eg in connect4 we have a sequence-based evaluation function, find our sequences, weight them and subtract that of opponent

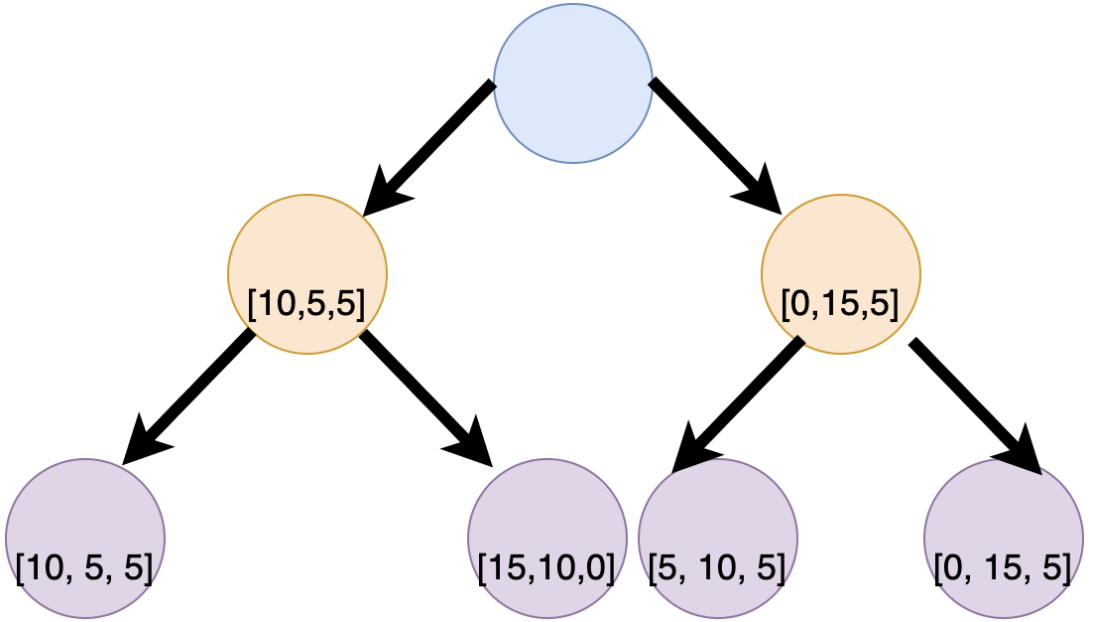
Dealing With 2-Player Limit: Max^n

- Max^n:
  - Each player is maximizing an evaluation
  - V_{p1} = eval(P1) - (eval(P2) + eval(P3) + …) / (n-1)
  - eval'(player) is the evaluation of that player from their prospective
  - Eg in connect4 we have a sequence-based evaluation function, find our sequences, weight them and subtract that of opponent
- Notice: This reduces to minimax when n=2
  - eval(P1) = eval'(P1) - eval'(P2)
  - eval(P2) = eval'(P2) - eval'(P1)

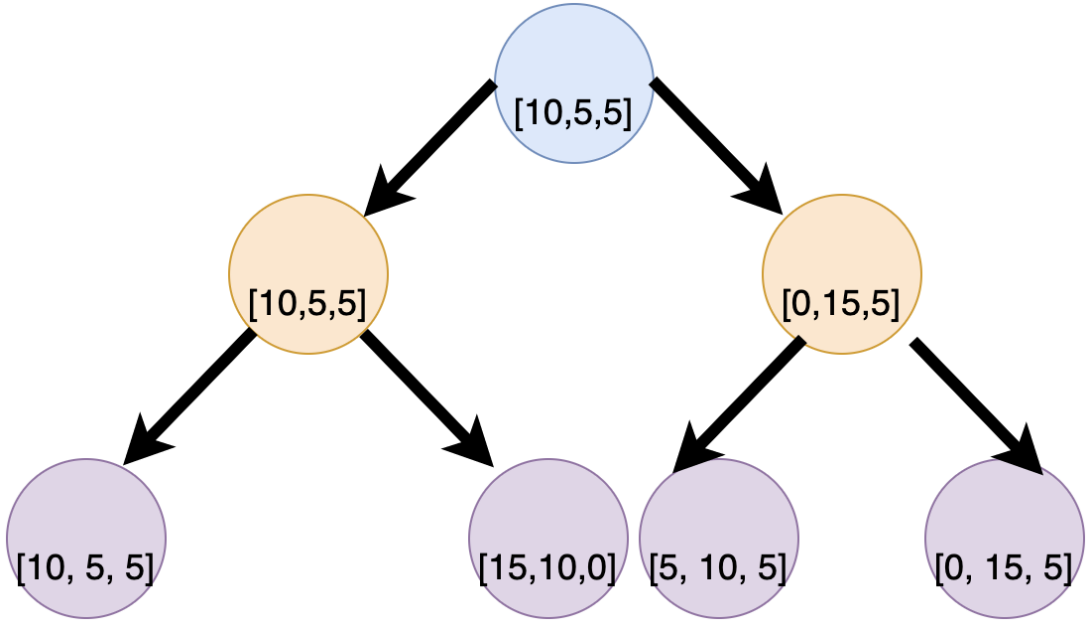
Max^n Example



Max^n Example

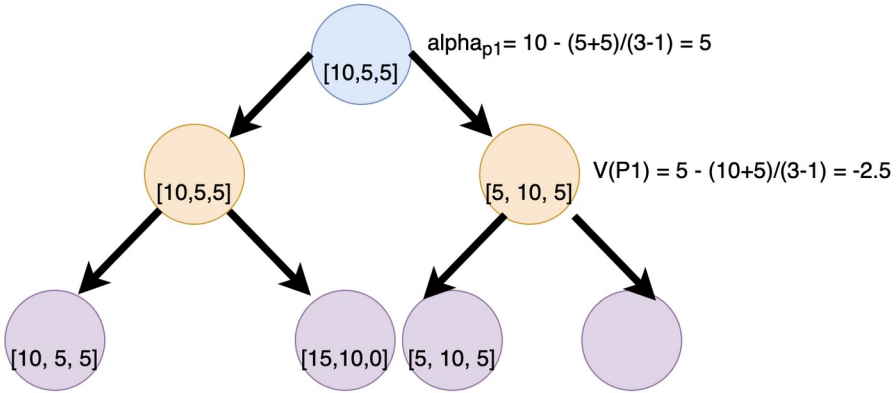


Max^n Example



Dealing With 2-Player Limit: Max^n

- Idea: Alphabeta pruning using Max^n:
  - eval can easily be rearranged to get to different perspectives
  - Prune if intermediate eval(P1) < eval(P1) for next state



Dealing With 2-Player Limit: Max^n

- Idea: Alphabeta pruning using Max^n:
  - eval can easily be rearranged to get to different perspectives
  - Prune if intermediate eval(P1) < eval(P1) for next state
- Issue: Opponent might find a move better for us but significantly worse for the opponent and chose that!

Dealing With 2-Player Limit: Max^n

- Idea: Alphabeta pruning using Max^n:
  - eval can easily be rearranged to get to different perspectives
  - Prune if intermediate eval(P1) < eval(P1) for next state
- Issue: Opponent might find a move better for us but significantly worse for the opponent and chose that!
- Solution: We have to make an upper bound on each element of the tuple:
  - (our score, opp1, opp2, …)
  - Let m = upper bound - our score

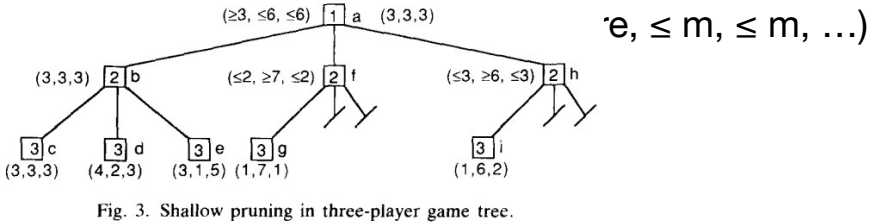


Fig. 3. Shallow pruning in three-player game tree.

e, ≤ m, ≤ m, …)

Korf et al 1991

Dealing With Stochastic Games: Expectimax

- A lot of games have elements of chance - eg dice, cards, random numbers, random events

Dealing With Stochastic Games: Expectimax

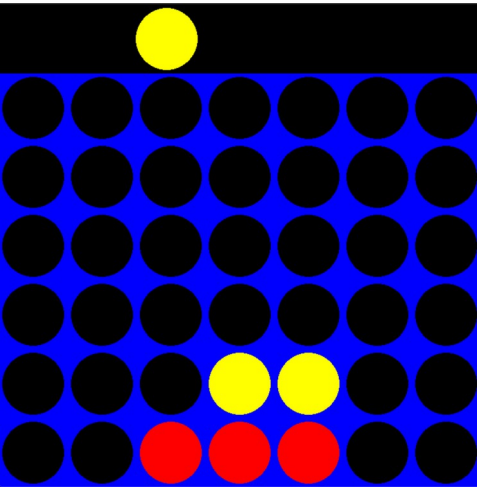
- A lot of games have elements of chance - eg dice, cards, random numbers, random events
- Key innovation over minimax: Chance nodes
    - Assume that the value of the chance node is equal to the expected value of its children

Dealing With Stochastic Games: Expectimax

- A lot of games have elements of chance - eg dice, cards, random numbers, random events
- Key innovation over minimax: Chance nodes
  - Assume that the value of the chance node is equal to the expected value of its children
    - Expected value:
      - V = 0
      - for each event i:
        - V += payout(i) * probability(i)
      - return V

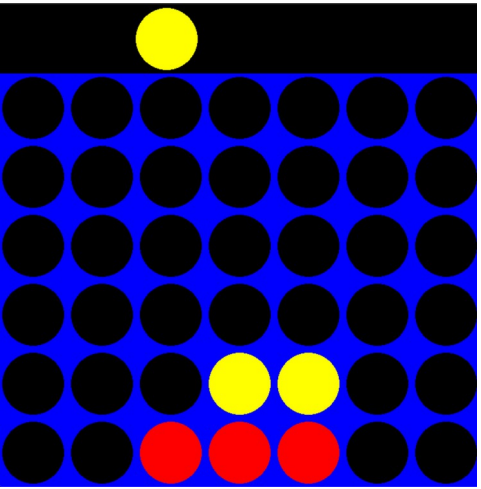
Benefits of Expectimax:

- How would minimax solve this problem?



Benefits of Expectimax:

- How would minimax solve this problem?
- Should we really just give up?



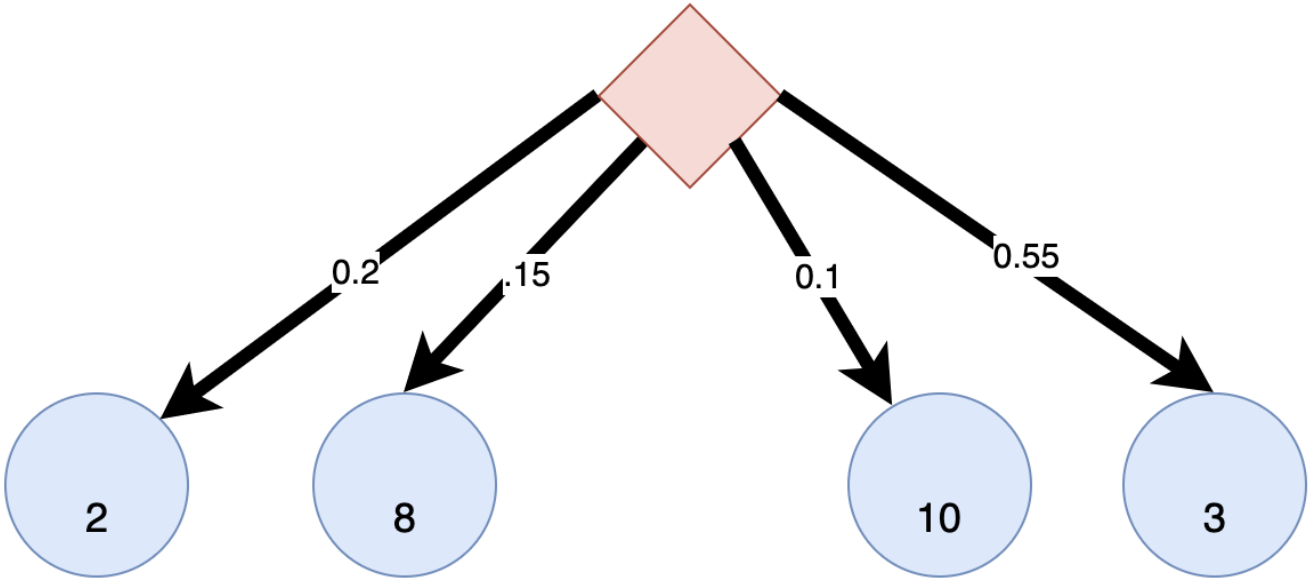
Minimax Algorithm

```
Def Max(state):
        if GameOver(state):
                return eval(state)
    value = -infinity
        for child in state:
                value = max(value, Min(child))
        return value

Def Min(state):
        if GameOver(state):
                return eval(state)
        value = infinity
        for child in state:
                value = min(value, Max(child))
        return value
```

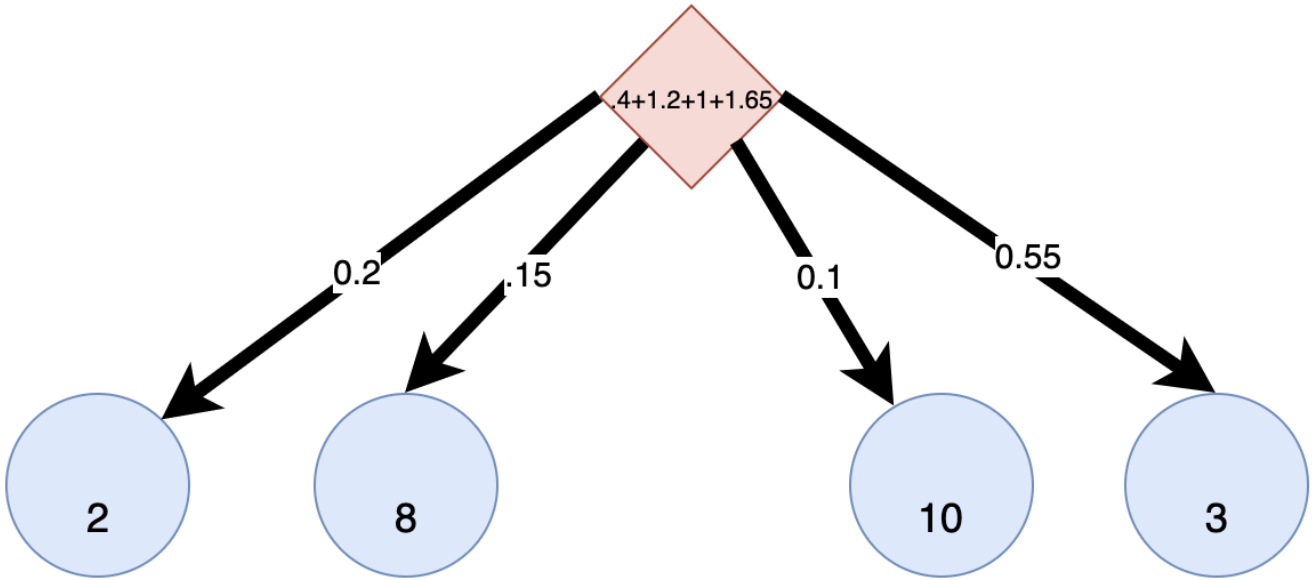
Expectimax Algorithm

```
Def Max(state):
        if GameOver(state):
                return eval(state)
    value = -infinity
        for child in state:
                value = max(value, Exp(child))
        return value

Def Exp(state):
        if GameOver(state):
                return eval(state)
        value = 0
        for child in state:
                value += probability(child) * Max(child)
        return value
```

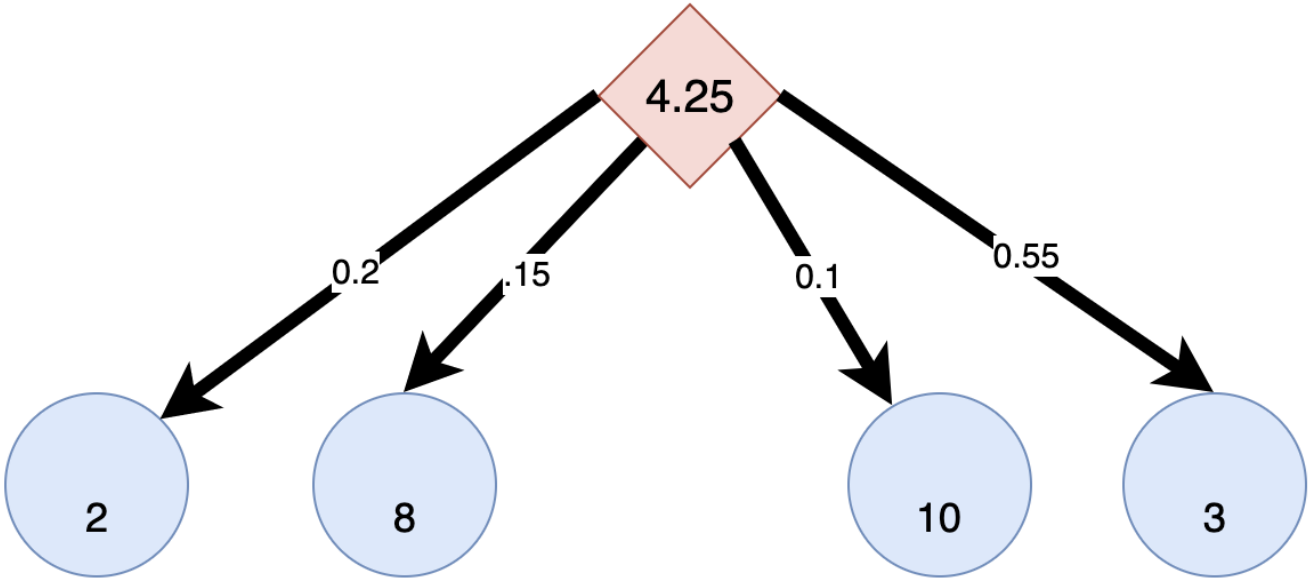
Exp Nodes



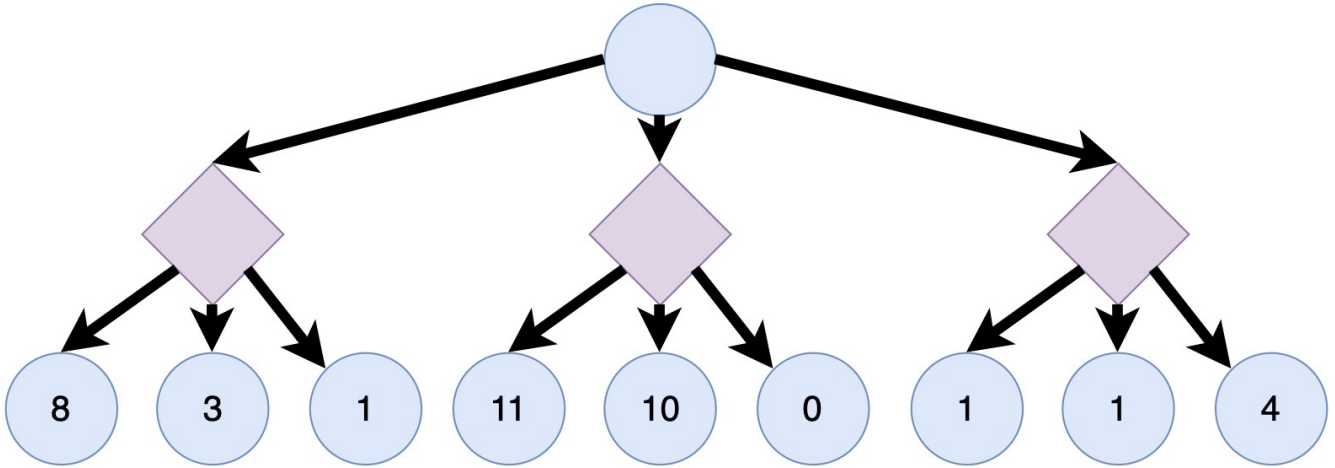
Exp Nodes



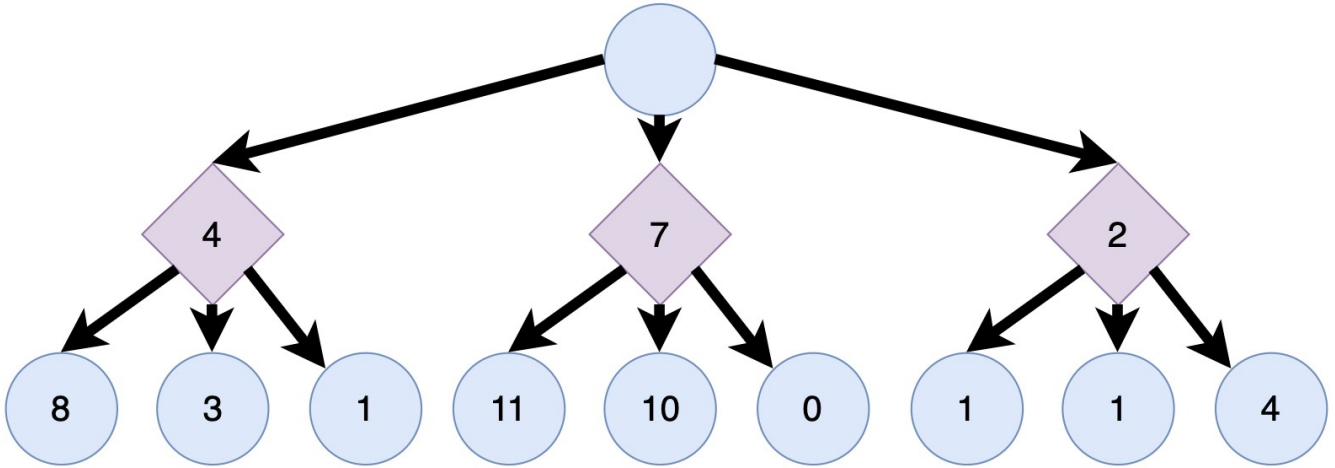
Exp Nodes



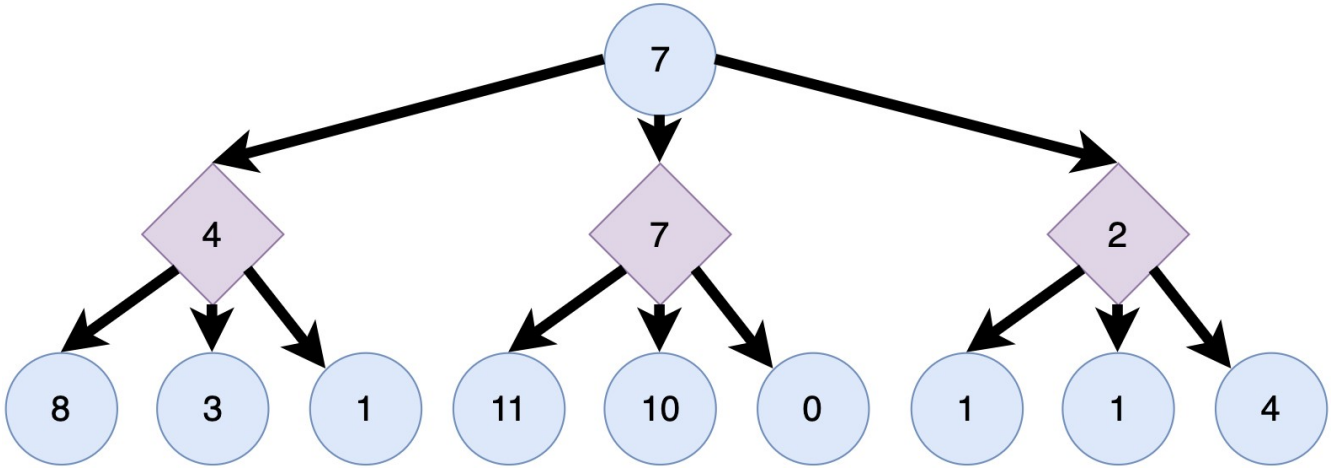
Expectimax in Action



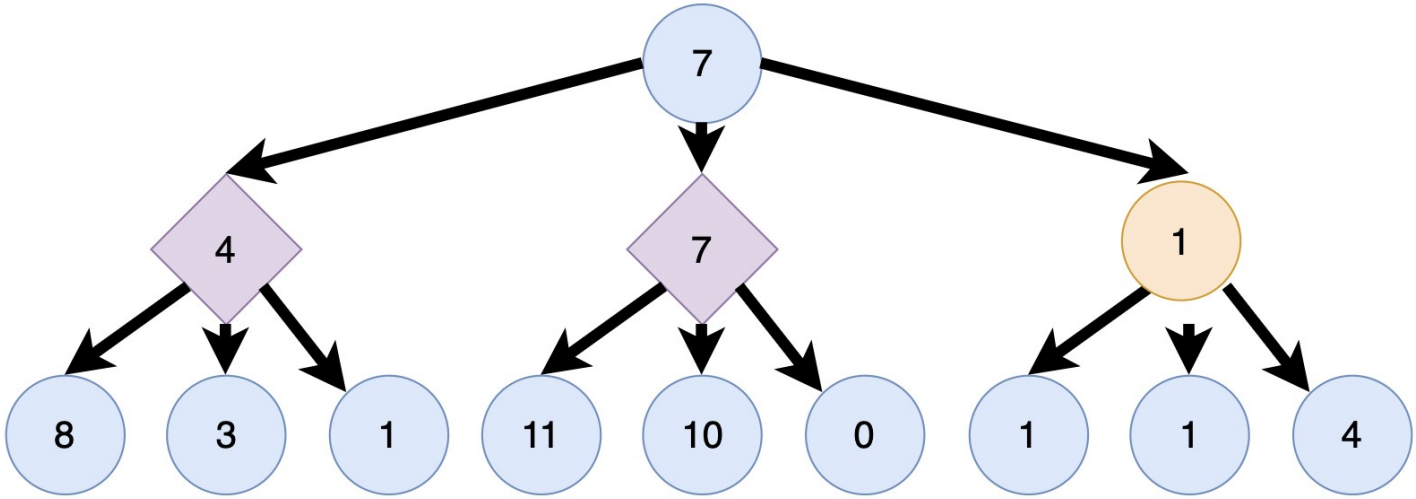
Expectimax in Action



Expectimax in Action



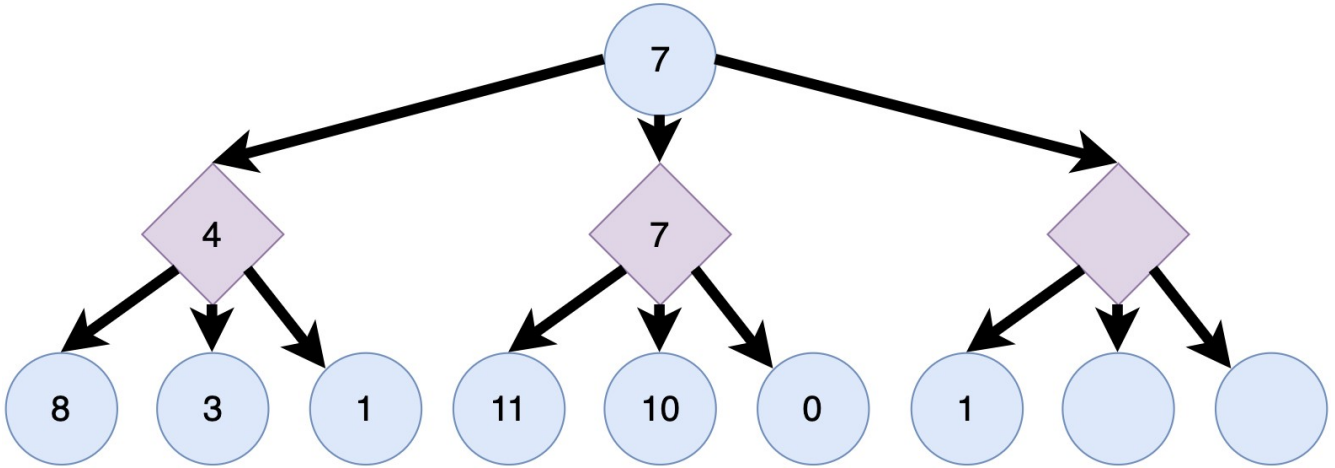
Exp, Max, Min



Exp, Max, Min

- Typical gameplay is Max -> Exp -> Min -> Exp -> Max -> …
- Examples:
  - Many “game night” style board games
    - Monopoly, Settlers of Catan
  - Skill & luck combination casino games:
    - Poker, backgammon

Pruning in Expectimax?



Pruning in Expectimax?

- Well, maybe
  - Need an upper or lower bound
  - Prune if:
    - Sum(children seen) + max bound * children unseen ≤ alpha
    - Sum(children seen) + max bound * children unseen ≥ beta

Discussion

- You are competing against a chaotic genius who plays the optimal move ⅔ of the time and a random move ⅓ of the time
- Which algorithm would you choose to defeat them?

Discussion

- You are competing against a chaotic genius who plays the optimal move ⅔ of the time and a random move ⅓ of the time
- Which algorithm would you choose to defeat them?
- What other information would influence your decision?

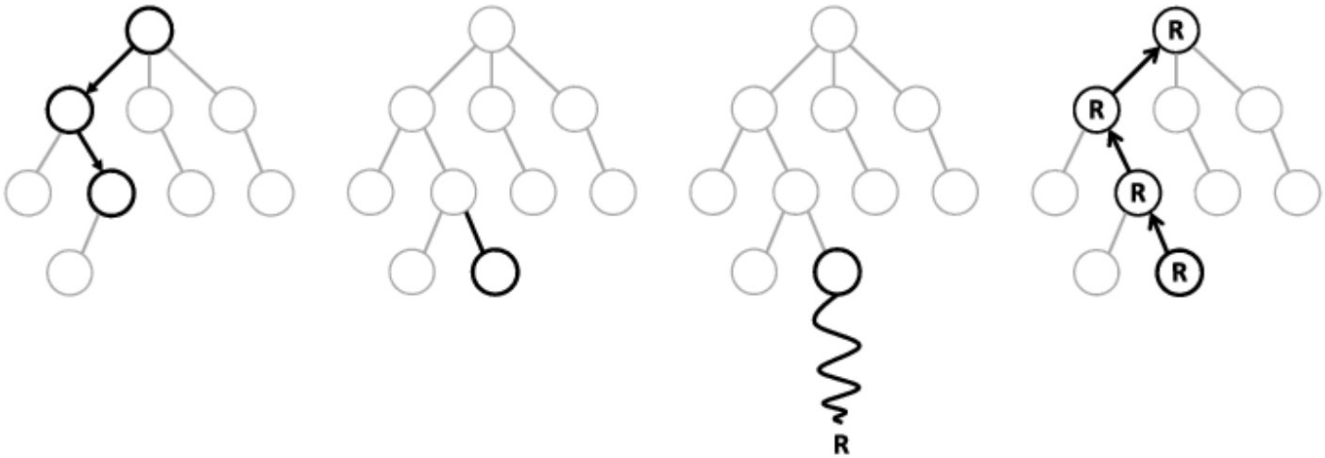
Discussion

- You are ~~competing against a chaotic genius~~ maneuvering a robot who ~~plays the optimal move~~ performs actions perfectly ⅔ of the time and ~~a random move~~ malfunctions ⅓ of the time
- Which algorithm would you choose to defeat them?
- What other information would influence your decision?

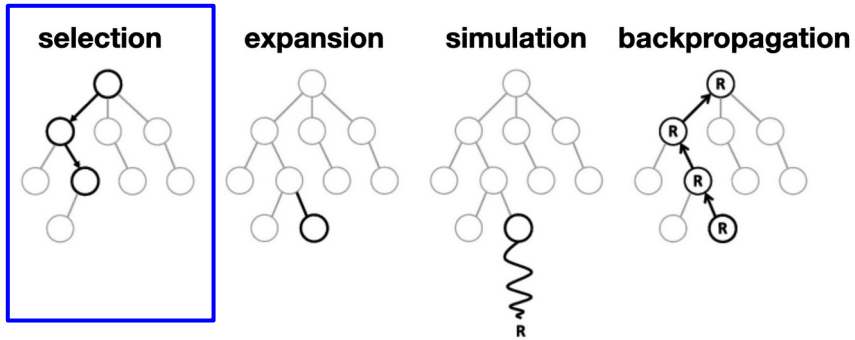
Lecture 4

Monte-Carlo Directed Searches

selection **expansion** **simulation** **backpropagation**

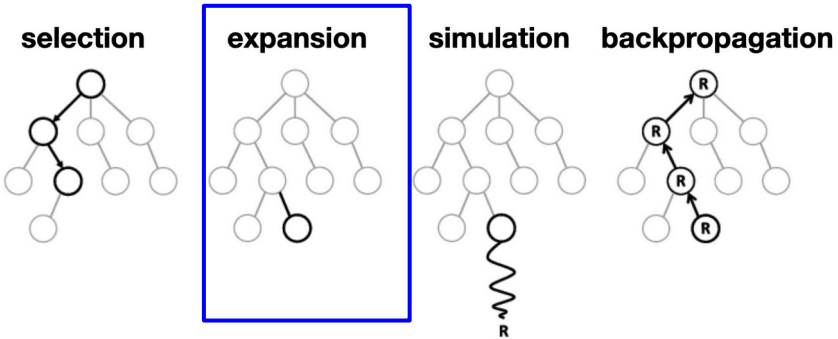


Monte-Carlo Directed Searches



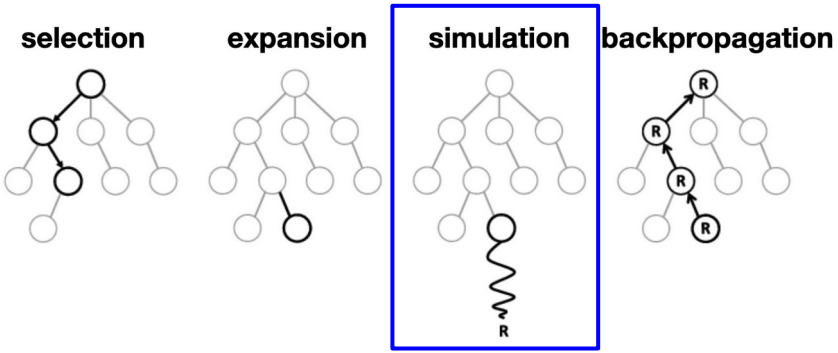
Select a node
to expand

Monte-Carlo Directed Searches



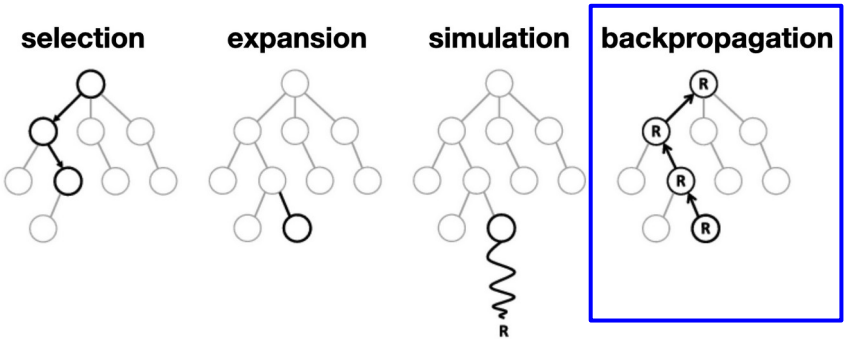
expand the node

Monte-Carlo Directed Searches



simulate random game(s) to a terminal node

Monte-Carlo Directed Searches



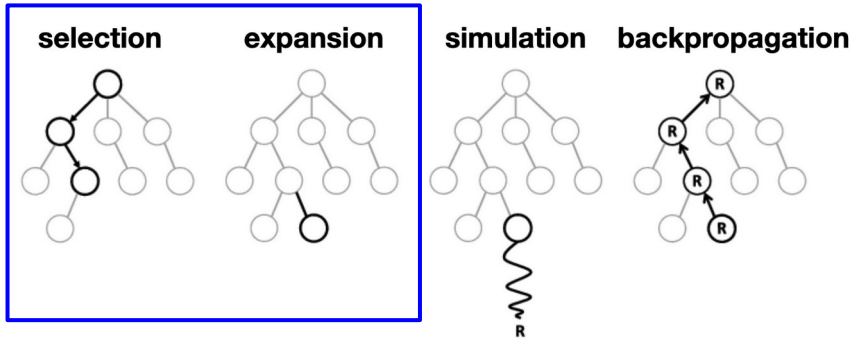
“backpropagate”

If that game went well, the preceding nodes are considered better

Monte-Carlo Directed Searches: Selection & Expansion

Basic idea: Select a node to expand based on two factors:

- How good the node looks given our previous expansions
- How often we have been there



UTC1 Selection criterion:

- w_i - wins from state i
- s_i - number of times state i played
- c - balance term
- s_p - number of times parent visited
- can also include an evaluation function

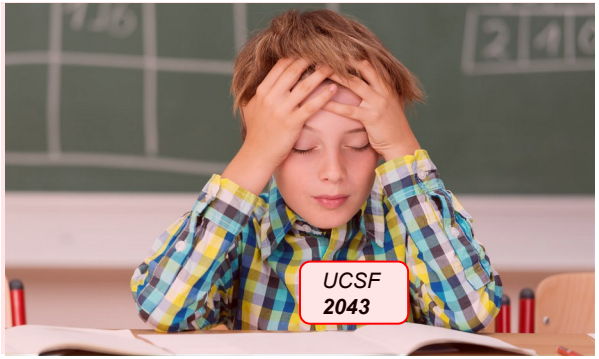
$$\frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

Should I switch majors?

New major might be more interesting…

More lucrative…

Need to learn new skills…



Already have foundational knowledge in current major…

Already made friends in current major…

Already have a professional network in current major…

“Do what you know”

is good advice…

“Don’t be afraid to try something new”

… is also good advice

the exploration vs. exploitation tradeoff:

explore something new

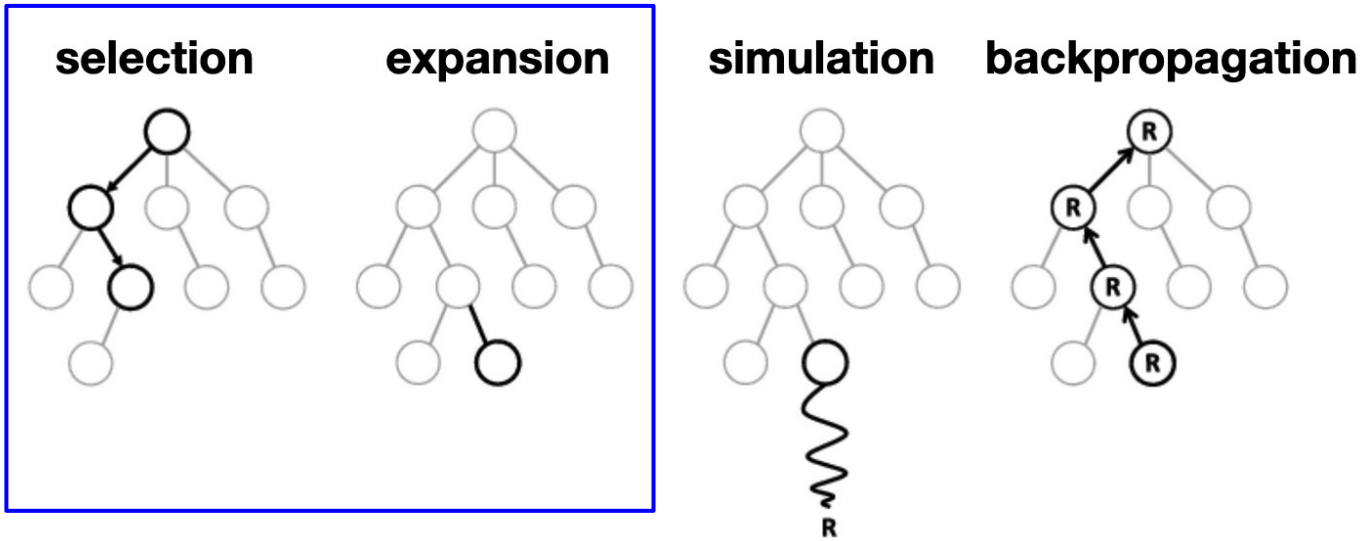
vs.

do the thing that already works

not just for pre-med first graders… or undergrads…

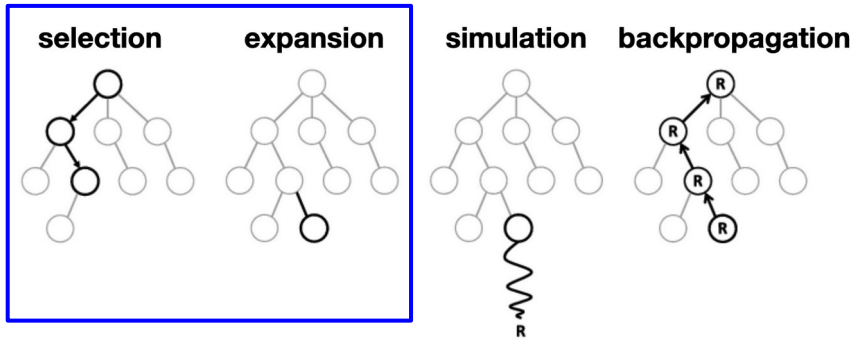
AIs and their developers struggle too

Monte-Carlo Directed Searches: Selection & Expansion



Choose which parts of the state space to explore via a **selection policy**

Monte-Carlo Directed Searches: Selection & Expansion



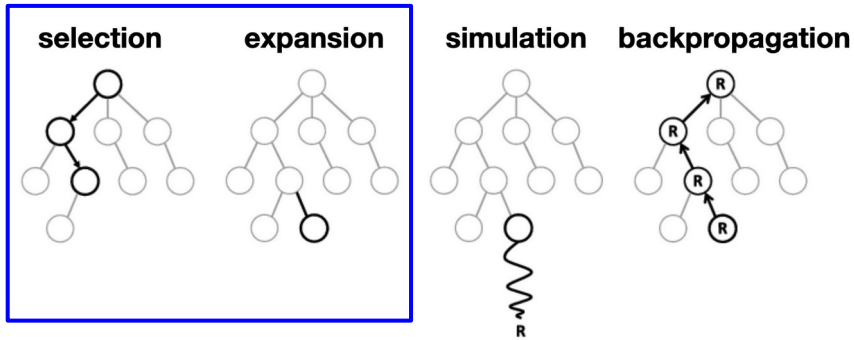
Pure Monte Carlo Selection

Select Nodes at Random

Pure Monte Carlo is…

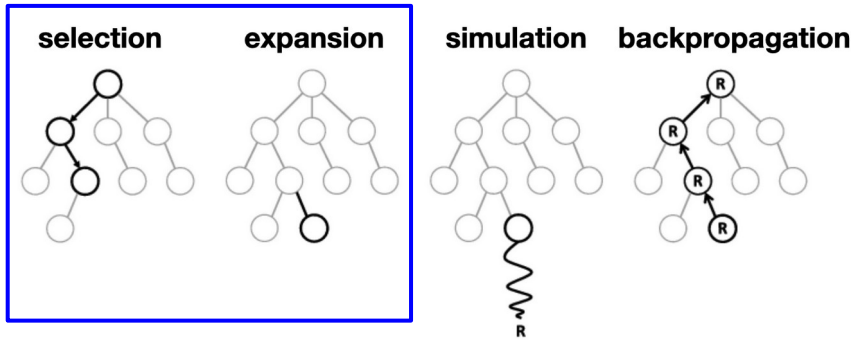
- Ok for some stochastic games
- Insufficient for most games

Monte-Carlo Directed Searches: Selection & Expansion



Need a way to direct expansion towards important parts of the game tree

Monte-Carlo Directed Searches: Selection & Expansion



state is good if…

we tend to win from this state (exploit)

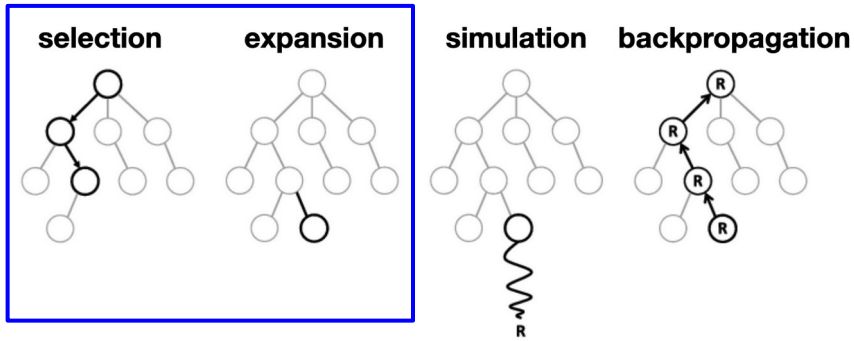
we don't often choose this state (explore)

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

***UTC* Selection criterion:**

- w_i - wins from state i
- s_i - number of times state i played
- c - balance term
- s_p - number of times parent visited

Monte-Carlo Directed Searches: Selection & Expansion



state is good if…

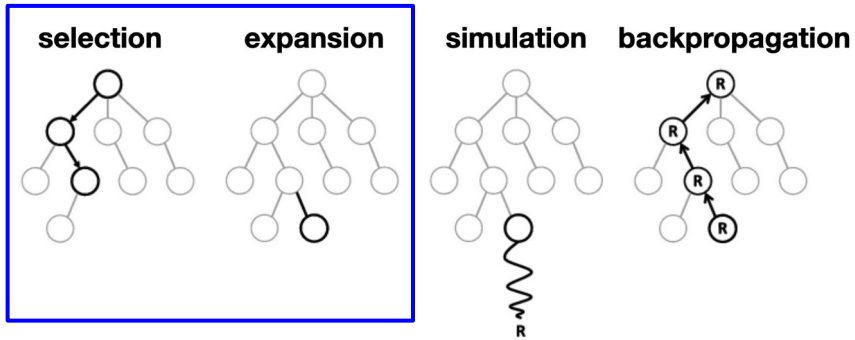
c manages the tradeoff

***UTC* Selection criterion:**

- w_i - wins from state i
- s_i - number of times state i played
- c - balance term
- s_p - number of times parent visited
- can also include an evaluation function

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

Monte-Carlo Directed Searches: Selection & Expansion



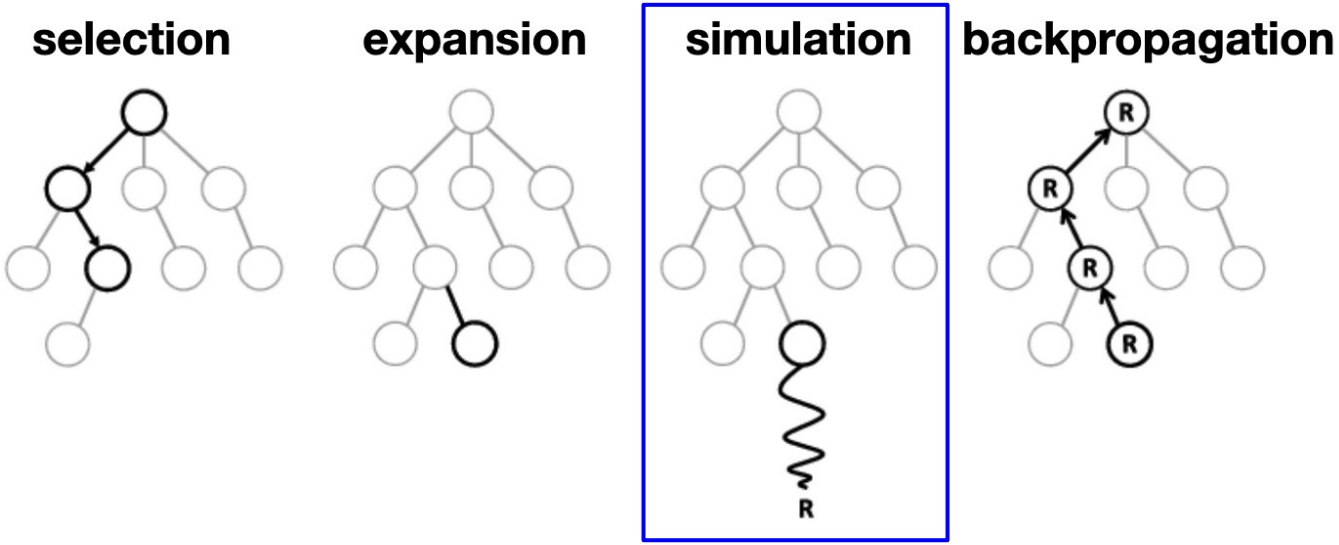
"Some programs use slightly different formulas; for example, AlphaZero adds in a term for move probability, which is calculated by a neural network trained from past self-play."

***UTC* Selection criterion:**

- w_i - wins from state i
- s_i - number of times state i played
- c - balance term
- s_p - number of times parent visited
- can also include an evaluation function

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

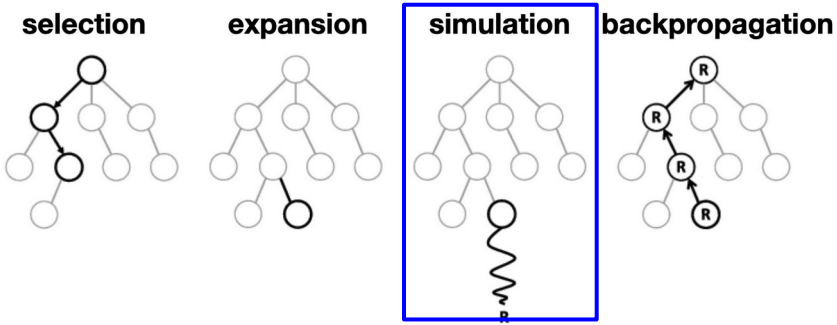
Monte-Carlo Directed Searches: Simulation



Simulate one or more games to a terminal node

Monte-Carlo Directed Searches: Simulation

- Simulate a game to a terminal node
- Observe which player won
- Repeat

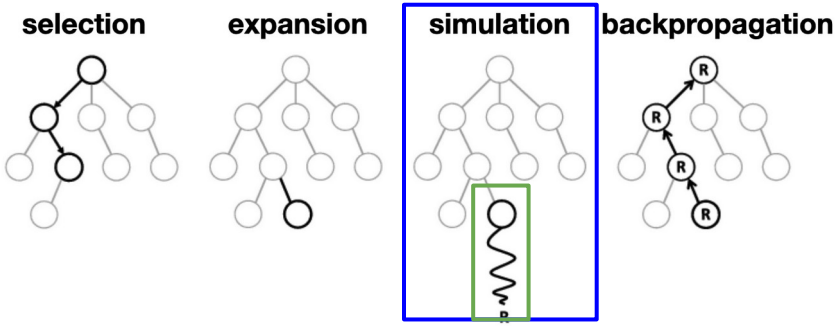


Monte-Carlo Directed Searches: Simulation

- Simulate a game to a terminal node
- Observe which player won
- Repeat

What kind of game? How do we model each player?

- Random moves?
- Intelligent moves?

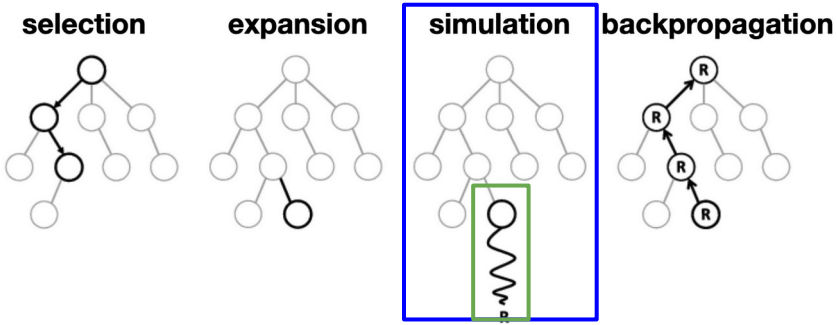


"playout" or "rollout"

Monte-Carlo Directed Searches: Simulation

Playout policy controls the simulation

- *“For Go and other games, playout policies have been successfully learned from self-play by using neural networks”*
- *“Sometimes game-specific heuristics are used, such as “consider capture moves” in Chess…”*

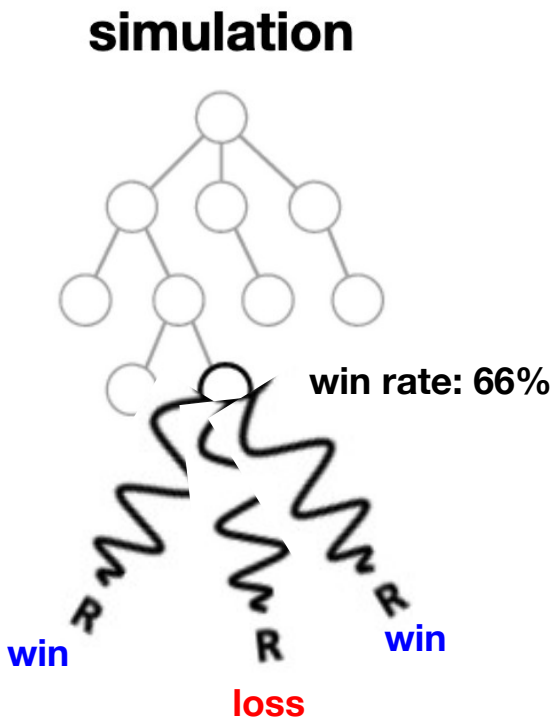


“playout” or “rollout”

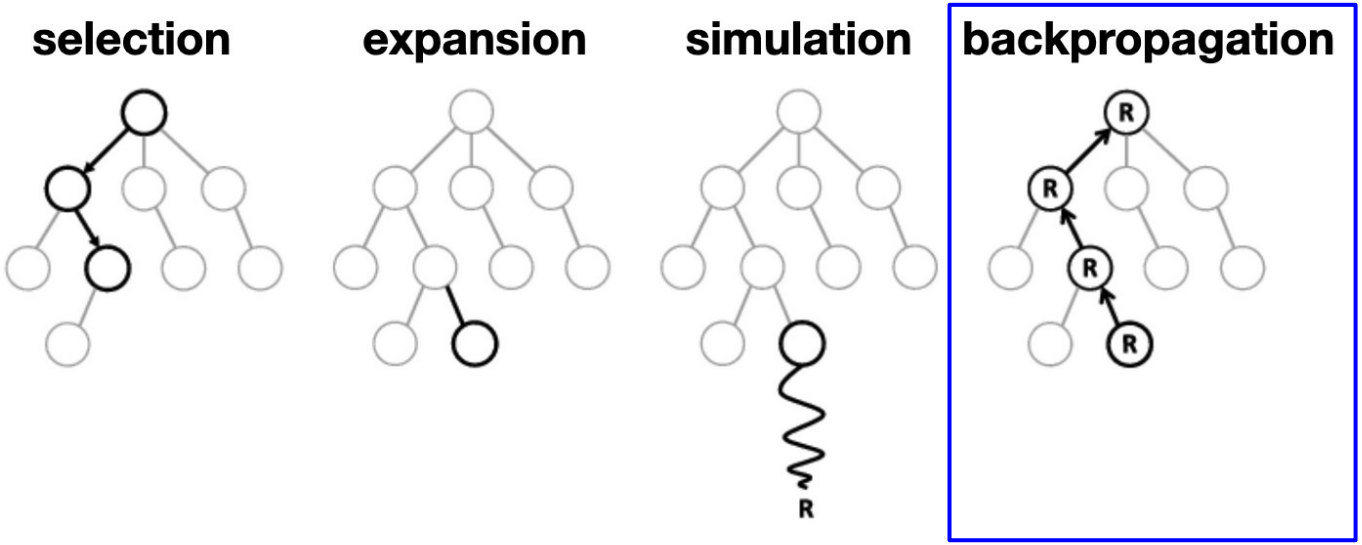
Monte-Carlo Directed Searches: Simulation

- Do N simulations starting from the current state of the game
- Track which moves get the highest win percentage

- Want to converge to optimal play as N increases



Monte-Carlo Directed Searches: Backpropagation



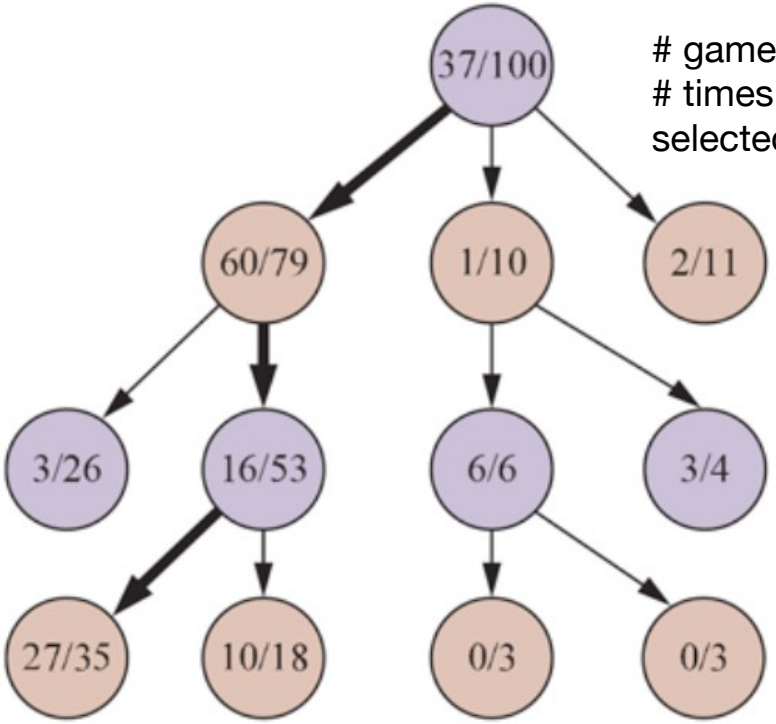
Update the score for preceding nodes

MCTS Example

Selection

Selection: UCT

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

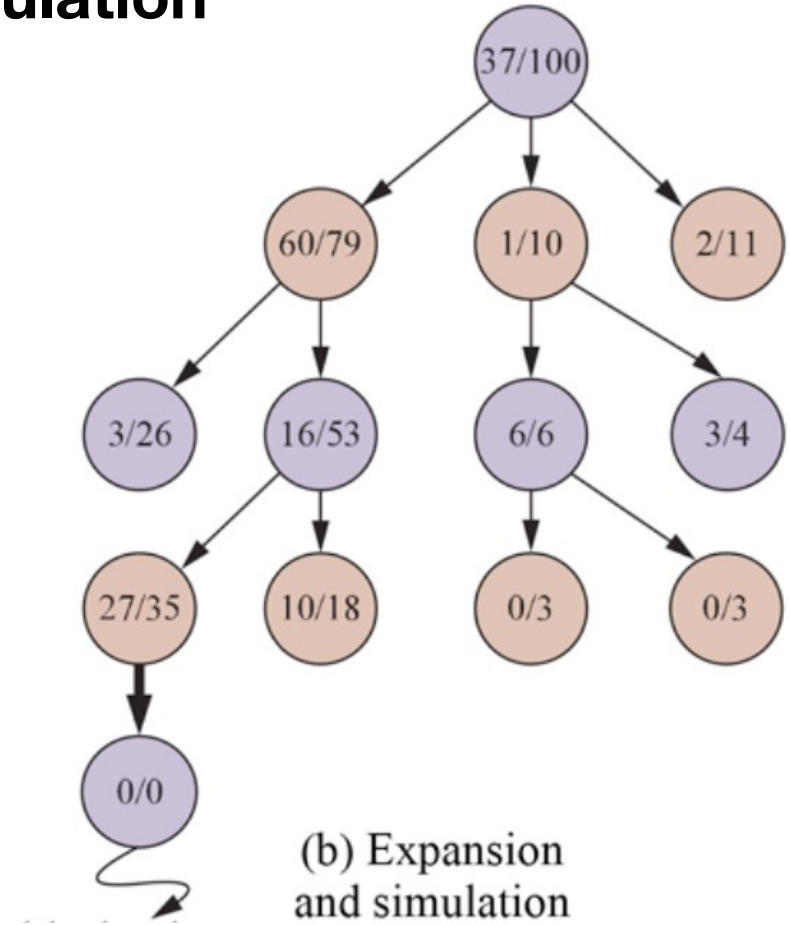


games won /
times this node was selected

Player 1

Player 2

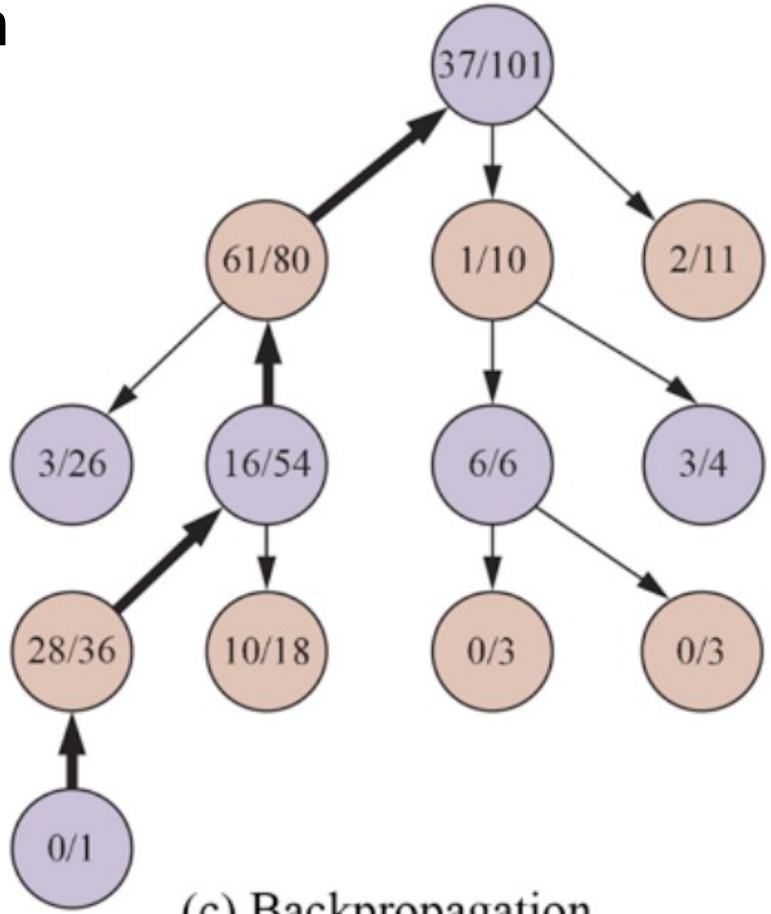
Expansion & Simulation



(b) Expansion and simulation

Orange wins

Backpropagation



(c) Backpropagation

MCTS Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

MCTS Complexity

Recall: Minimax and Alpha-Beta pruning are exponential in depth

MCTS is linear in depth for a single playout

(assuming a constant-time selection policy)

Plenty of time for multiple playouts…

keep searching until we run out of time for our turn

Benefits of MCTS

- Scalable to high branching factor problems
- Always has a best move
- In practice:
  - MCTS tends to be better with large branching factor, large depth problems
  - alpha-beta is competitive in smaller problems
- Can also be used in a non-adversarial setting
- Can be used in non-deterministic games and games without perfect information as long as we can simulate them

Discussion

- How would we apply adversarial search to something like:
- Consider:
  - Actions/Branching factor
  - Designing evaluation function
  - Use of variants

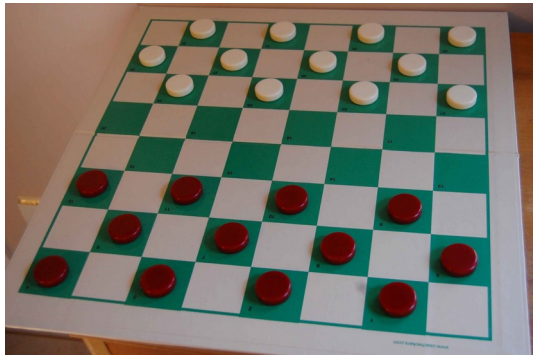
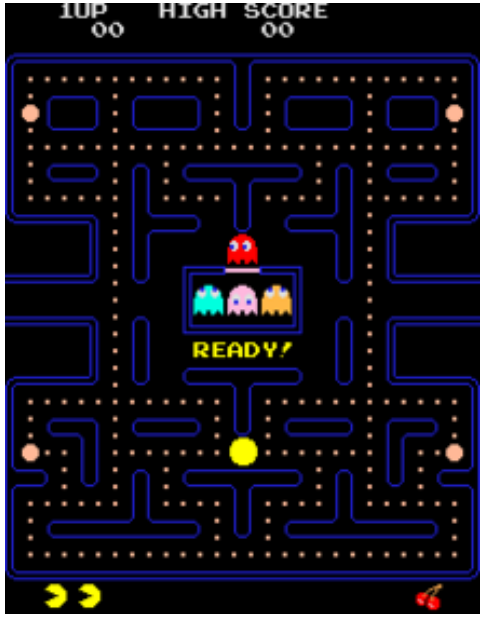


Photo credit: Jud McCranie

Discussion

- How would we apply adversarial search to something like:
- Consider:
  - Actions/Branching factor
  - Designing evaluation function
    - Short
    - Long
    - From monster's perspective (designing AI for game) and from ours
  - Use of variants



Discussion

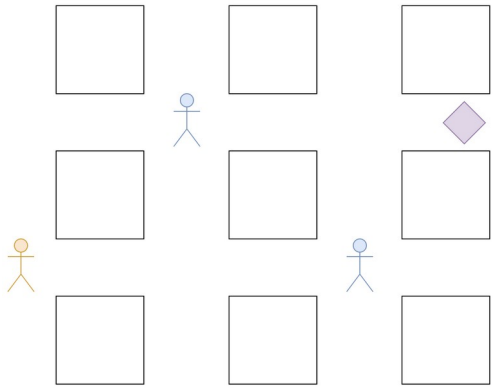
- In stealth games & heist movies, we want the characters to be able to slip past the detection system

Discussion

- In stealth games & heist movies, we want the characters to be able to slip past the detection system
- In the real world, not so much

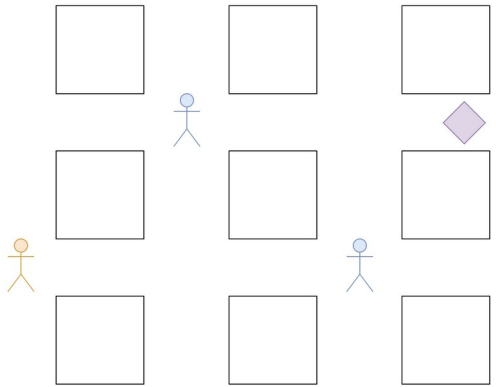
Discussion

- In stealth games & heist movies, we want the characters to be able to slip past the detection system
- In the real world, not so much
- A “security game” is a simulation in which the adversary tries to outsmart our security system

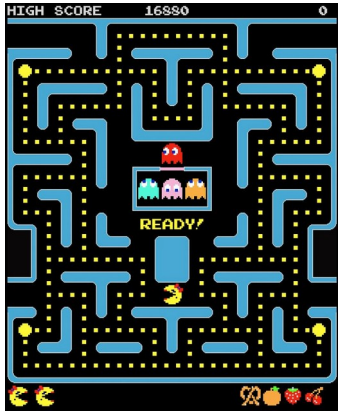


Discussion

- How would we design an algorithm to work with this?
  - What would the actions/branching factor be?
  - Does this violate any of the aforementioned constraints?
  - Which algorithm would we use to solve this?
    - If minimax, how would we make an evaluation function
    - If MCTS, why?



Real World Applications

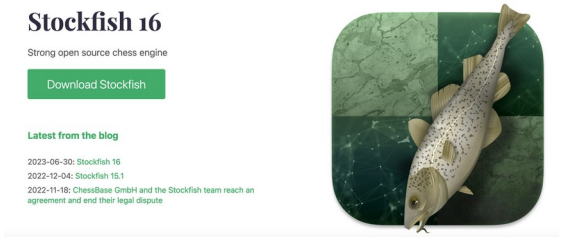


Games:

- Mrs. Pacman
- Total War
- Computer Chess Champion Stockfish

Security Games:

- We have a security system to handle attacks
- Security patrols vs criminal attackers
  - Analysis by the US Coast Guard
  - Analysis of Terrorist Threats by LAX
  - Evaluating Security Threats On Biometric Data



Chemical Synthesis

Recap

Informed Search: How to direct a search path between our current state and the goal when we have full control

Adversarial Search: How to take action to lead us to a goal from our current state when we have a rational adversary

Limitations

- We can’t work in a continuous action space
- We can’t “think” for ourselves
- Our methods are expensive