CS486/686: Introduction to Artificial Intelligence
Lecture 3b - Informed/Heuristic Search

Jesse Hoey & Victor Zhong

School of Computer Science, University of Waterloo

January 14, 2025

Readings: Poole & Mackworth Chap. 3.6–3.8, 14.3

Heuristic Search

- **Idea:** don't ignore the goal when selecting paths
- Often there is extra knowledge that can be used to guide the search: **heuristics**

Heuristic Search

- **Idea:** don't ignore the goal when selecting paths
- Often there is extra knowledge that can be used to guide the search: **heuristics**
- $h(n)$ is an **estimate** of the cost of the **shortest path** from node n to a goal node

Heuristic Search

- **Idea:** don't ignore the goal when selecting paths
- Often there is extra knowledge that can be used to guide the search: **heuristics**
- $h(n)$ is an **estimate** of the cost of the **shortest path** from node n to a goal node
- $h(n)$ uses only **readily obtainable** information (that is easy to compute) about a node
- computing the heuristic must be **much easier** than solving the problem

Heuristic Search

- **Idea:** don't ignore the goal when selecting paths
- Often there is extra knowledge that can be used to guide the search: **heuristics**
- $h(n)$ is an **estimate** of the cost of the **shortest path** from node n to a goal node
- $h(n)$ uses only **readily obtainable** information (that is easy to compute) about a node
- computing the heuristic must be **much easier** than solving the problem
- h can be **extended** to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$
- $h(n)$ is an **underestimate** if there is no path from n to a goal that has path length less than $h(n)$

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the **straight-line distance** from n to the closest goal as the value of $h(n)$

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the **straight-line distance** from n to the closest goal as the value of $h(n)$
- If the nodes are locations and cost is **time**, we can use the distance to a goal divided by the maximum speed

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the **straight-line distance** from n to the closest goal as the value of $h(n)$
- If the nodes are locations and cost is **time**, we can use the distance to a goal divided by the maximum speed
- If nodes are locations on a grid and cost is distance, we can use the **Manhattan distance**: distance by taking horizontal and vertical moves only

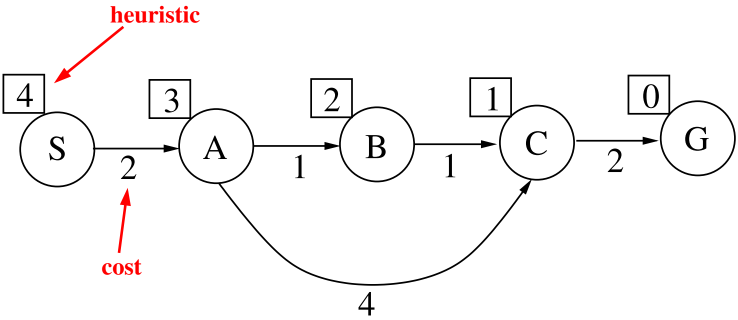
Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the **straight-line distance** from n to the closest goal as the value of $h(n)$
- If the nodes are locations and cost is **time**, we can use the distance to a goal divided by the maximum speed
- If nodes are locations on a grid and cost is distance, we can use the **Manhattan distance**: distance by taking horizontal and vertical moves only
- Think of heuristics for your favorite games: chess? go? starcraft?

Greedy Best-First Search

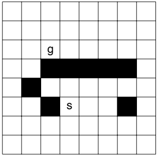
- **Idea:** select the path whose end is **closest to a goal** according to the **heuristic** function
- **Best-first search** selects a path on the frontier with **minimal h-value**
- It treats the frontier as a **priority queue** ordered by h

Illustrative Example: Best-First Search



Best first: S-A-C-G (not optimal)

Graph Search Algorithm with Multiple Path Pruning



- Use **best-first search** to get from **s** to **g**
- Number the nodes as they are removed
- Use multiple path pruning
- break ties arbitrarily
- Use the **Manhattan distance** as heuristic

Input: a graph with start nodes, **Boolean procedure** $goal(n)$ that tests if n is a goal node
$frontier \leftarrow \{\langle s \rangle : s \text{ is a start node}\}$
$explored \leftarrow \{\}$
while *frontier* is not empty **do**
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
  if $n_k \notin explored$ **then**
    add n_k **to** *explored*
    if $goal(n_k)$ **then**
      return $\langle n_0, \ldots, n_k \rangle$
    for each neighbor n of n_k **do**
      add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*

Properties of GBFS

- Space and Time Complexities

Properties of GBFS

- Space and Time Complexities
  Both complexities are exponential

Properties of GBFS

- Space and Time Complexities
  Both complexities are exponential
- Completeness and Optimality

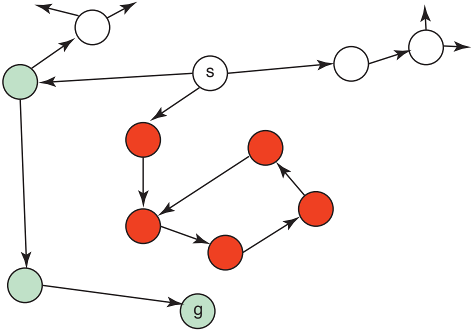
Properties of GBFS

- Space and Time Complexities
  Both complexities are exponential
- Completeness and Optimality
  No, GBFS is not complete. It could be stuck in a cycle
  No, GBFS is not optimal. GBFS may return a sub-optimal path first

Heuristic Depth-First Search

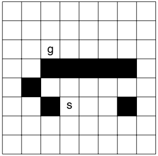
- **Idea:** Do a **depth-first** search, but add paths to the stack **ordered according to** h
- **Locally** does a best-first search, but aggressively pursues the **best looking** path (even if it ends up being worse than one higher up)
- Same asymptotic properties (and problems) as depth-first search
- Is often **used in practice**

Illustrative Graph: Heuristic Search



Cost of an arc is its length
Heuristic: euclidean distance
Red nodes all look better than green nodes
A challenge for heuristic depth first search

Graph Search Algorithm with Multiple Path Pruning



- Use **heuristic depth-first search**
- Number the nodes as they are removed
- Use multiple path pruning
- Break ties arbitrarily
- Use **Manhattan distance** as heuristic

Input: Graph G, start nodes S, Boolean procedure $goal(n)$ that tests if n is a goal node

$frontier \leftarrow \{(s) : s \text{ is a start node}\}$
$explored \leftarrow \{\}$
while *frontier* is not empty **do**
  select and **remove** path $(n_0, \ldots, n_k)$ from *frontier*
  if $n_k \notin explored$ **then**
    add n_k **to** *explored*
    if $goal(n_k)$ **then**
      return $(n_0, \ldots, n_k)$
    for each neighbors n of n_k **do**
      add $(n_0, \ldots, n_k, n)$ to *frontier*

A^* Search

- A^* search uses both path **cost and heuristic** values

A^* Search

- A^* search uses both path **cost and heuristic** values
- $cost(p)$: the cost of path p
- $h(p)$ estimates the cost from the end of p to a goal

A^* Search

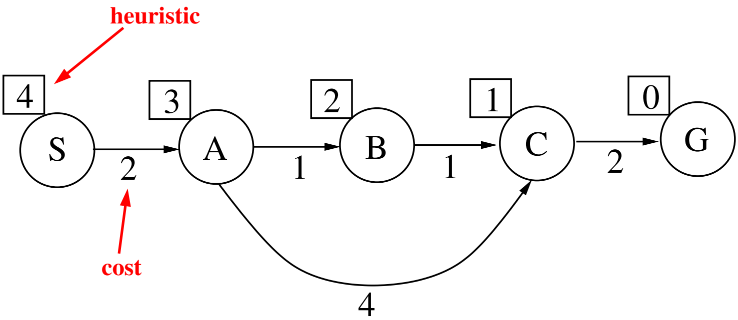
- A^* search uses both path **cost and heuristic** values
- $cost(p)$: the cost of path p
- $h(p)$ estimates the cost from the end of p to a goal
- Let $f(p) = cost(p) + h(p)$; $f(p)$ estimates the **total path cost** of going from a start node to a goal via p

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

A^* Search Algorithm

- A^* is a **mix** of **lowest-cost-first** and **best-first search**
- It treats the frontier as a **priority queue ordered by** $f(p)$
- It always selects the node on the frontier with the **lowest estimated distance** from the start to a goal node constrained to go via that node

Illustrative Example: A^* search



Recall best first: S-A-C-G (not optimal)
A^* : S-A-B-C-G (optimal)

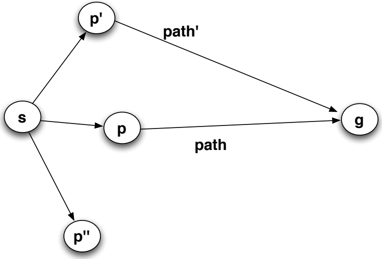
Admissibility of A^*

If there is a solution, A^* always finds an **optimal** solution—the **first** path to a goal selected—if

- The branching factor is **finite**
- Arc costs are **bounded above zero** (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ)
- $h(n)$ is a **lower bound** on the length (cost) of the shortest path from n to a goal node

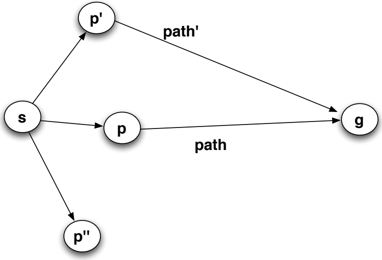
Admissible heuristics never overestimate the cost to the goal

Why is A^* with admissible h optimal?



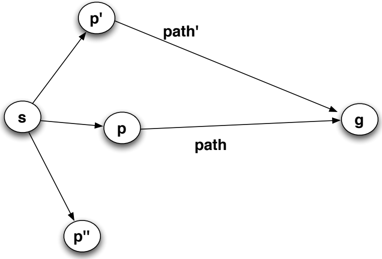
- Assume: *path* $s \rightarrow p \rightarrow g$ is the optimal
- $f(p) = cost(s, p) + h(p) < cost(s, g)$ due to h being a lower bound

Why is A^* with admissible h optimal?



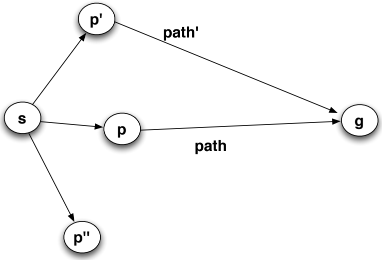
- Assume: *path* $s \to p \to g$ is the optimal
- $f(p) = cost(s, p) + h(p) < cost(s, g)$ due to h being a lower bound
- $cost(s, g) < cost(s, p') + cost(p', g)$ due to optimality of *path*

Why is A^* with admissible h optimal?



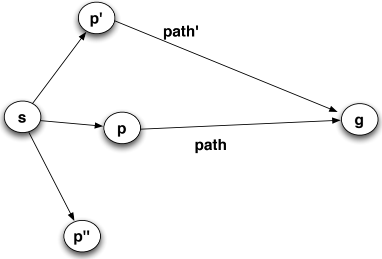
- Assume: *path* $s \to p \to g$ is the optimal
- $f(p) = cost(s,p) + h(p) < cost(s,g)$ due to h being a lower bound
- $cost(s,g) < cost(s,p') + cost(p',g)$ due to optimality of *path*
- Therefore $cost(s,p) + h(p) = f(p) < cost(s,p') + cost(p',g)$

Why is A^* with admissible h optimal?



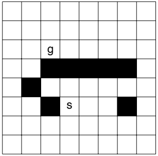
- Assume: *path* $s \rightarrow p \rightarrow g$ is the optimal
- $f(p) = cost(s,p) + h(p) < cost(s,g)$ due to h being a lower bound
- $cost(s,g) < cost(s,p') + cost(p',g)$ due to optimality of *path*
- Therefore $cost(s,p) + h(p) = f(p) < cost(s,p') + cost(p',g)$
- Therefore, we will never choose *path*$'$ while *path* is unexplored

Why is A^* with admissible h optimal?



- Assume: *path* $s \rightarrow p \rightarrow g$ is the optimal
- $f(p) = cost(s, p) + h(p) < cost(s, g)$ due to h being a lower bound
- $cost(s, g) < cost(s, p') + cost(p', g)$ due to optimality of *path*
- Therefore $cost(s, p) + h(p) = f(p) < cost(s, p') + cost(p', g)$
- Therefore, we will never choose $path'$ while *path* is unexplored
- A^* halts, as the costs of the paths on the frontier keeps increasing and will eventually exceed any finite number

Graph Search Algorithm with Multiple Path Pruning



- Use **A* search**
- Number the nodes as they are removed
- Use multiple path pruning
- break ties arbitrarily
- Use **Manhattan distance** as heuristic

Input: Graph G, start nodes S, Boolean procedure $goal(n)$ that tests if n is a goal node

frontier $\leftarrow \{(s) : s \text{ is a start node}\}$
explored $\leftarrow \{\}$
while *frontier* is not empty **do**
  select and **remove** path $(n_0, \ldots, n_k)$ from *frontier*
  if $n_k \notin explored$ **then**
    add n_k **to** *explored*
    if $goal(n_k)$ **then**
      return $(n_0, \ldots, n_k)$
    for each neighbors n of n_k **do**
      add $(n_0, \ldots, n_k, n)$ to *frontier*

Properties of A* Search

- Space and Time Complexities

Properties of A* Search

- Space and Time Complexities
  Both complexities are exponential

Properties of A* Search

- Space and Time Complexities
  Both complexities are exponential
- Completeness and Optimality

Properties of A* Search

- Space and Time Complexities
  Both complexities are exponential
- Completeness and Optimality
  Yes and Yes, (assuming the heuristic function is admissible, the branching factor is finite, and arc costs are bounded above zero)

A* is Optimally Efficient

Among all optimal algorithms that start from the same start node and use the same heuristic, A* expands the fewest nodes

A* is Optimally Efficient

Among all optimal algorithms that start from the same start node and use the same heuristic, A* expands the fewest nodes

- No algorithm with the same information can do better

A* is Optimally Efficient

Among all optimal algorithms that start from the same start node and use the same heuristic, A* expands the fewest nodes

- No algorithm with the same information can do better
- A* expands the minimum number of nodes to find the optimal solution

A* is Optimally Efficient

Among all optimal algorithms that start from the same start node and use the same heuristic, A* expands the fewest nodes

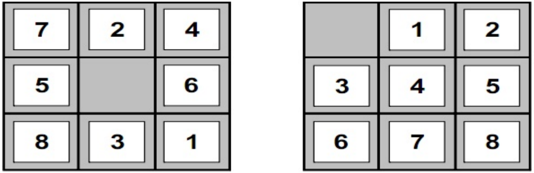
- No algorithm with the same information can do better
- A* expands the minimum number of nodes to find the optimal solution
- Intuition for proof: any algorithm that does not expand all nodes with $f(n) < cost(s, g)$ run the risk of missing the optimal solution

Constructing an Admissible Heuristic

1. **Define a relaxed problem** by simplifying or removing constraints on the original problem
2. **Solve the relaxed problem** without search
3. The cost of the optimal solution to the relaxed problem is an **admissible heuristic** for the original problem

Constructing a Heuristic for the 8-Puzzle

8-Puzzle tiles can move into adjacent empty slot only



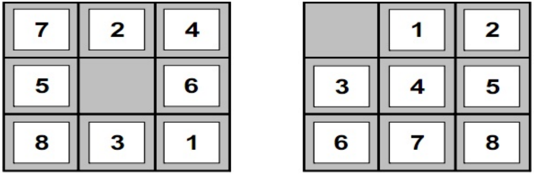
Start State Goal State

How can we relax the game (make it simpler, easier)?

1. Can move tile from position A to position B if A is next to B (ignore whether or not position is blank)
2. Can move tile from position A to position B if B is blank (ignore adjacency)
3. Can move tile from position A to position B

Constructing a Heuristic for the 8-Puzzle

8-Puzzle tiles can move into adjacent empty slot only



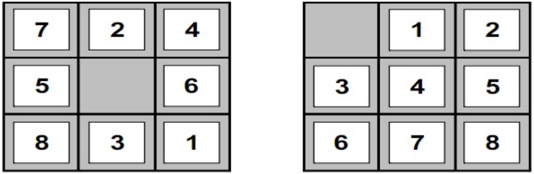
Start State **Goal State**

How can we relax the game (make it simpler, easier)?

1. Can move tile from position A to position B if A is next to B (ignore whether or not position is blank)
   - Leads to **Manhattan distance heuristic**
   - To solve the puzzle need to slide each tile into its final position
   - **Admissible**

Constructing a Heuristic for the 8-Puzzle

8-Puzzle tiles can move into adjacent empty slot only



Start State

Goal State

How can we relax the game (make it simpler, easier)?

3. Can move tile from position A to position B
   - leads to **misplaced tile heuristic**
   - To solve this problem need to move each tile into its final position
   - Number of moves = number of misplaced tiles
   - **Admissible**

Desirable Heuristic Properties

- We want a heuristic to be admissible
  - A* is optimal

Desirable Heuristic Properties

- We want a heuristic to be admissible
  - A* is optimal
- We want a heuristic to have higher values (close to h^*)
  - The closer h is to h^*, the more accurate h is

Desirable Heuristic Properties

- We want a heuristic to be admissible
  - A* is optimal
- We want a heuristic to have higher values (close to h^*)
  - The closer h is to h^*, the more accurate h is
- Prefer a heuristic that is very different for different states
  - h should help us choose among different paths
    If h is close to constant, it's not useful

Dominating Heuristic

Definition (dominating heuristic)

Given heuristics $h_1(n)$ and $h_2(n)$. $h_2(n)$ dominates $h_1(n)$ if

- $(\forall n \ (h_2(n) \geq h_1(n)))$.
- $(\exists n \ (h_2(n) > h_1(n)))$.

Dominating Heuristic

Definition (dominating heuristic)

Given heuristics $h_1(n)$ and $h_2(n)$. $h_2(n)$ dominates $h_1(n)$ if

- $(\forall n\ (h_2(n) \geq h_1(n)))$.
- $(\exists n\ (h_2(n) > h_1(n)))$.

Theorem

If $h_2(n)$ dominates $h_1(n)$, A using h_2 will never expand more nodes than A* using h_1.*

Which Heuristic of 8-puzzle is Better?

Which of the two heuristics of the 8-puzzle is better?

1. The Manhattan distance heuristic
2. The Misplaced tile heuristic

Which Heuristic of 8-puzzle is Better?

Which of the two heuristics of the 8-puzzle is better?

1. The Manhattan distance heuristic
2. The Misplaced tile heuristic

Manhattan distance is a better heuristic because it dominates the Misplaced tile heuristic

Multiple-Path Pruning & Optimal Solutions

Problem: What if a subsequent path to *n* is shorter than the first path to *n*?

- **Remove** all paths from the frontier that use the longer path
- **Change** the initial segment of the paths on the frontier to use the shorter path
- **Ensure this doesn't happen**: make sure that the shortest path to a node is found first (lowest-cost-first search)

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a shorter path to n; and suppose this shorter path is via path p' on the frontier
- Suppose path p' ends at node n'
- $cost(p) + h(n) \leq cost(p') + h(n')$ because p was selected before p'
- $cost(p') + cost(n', n) < cost(p)$ because the path to n via p' is shorter (by assumption)

$$cost(n', n) < cost(p) - cost(p') \leq h(n') - h(n)$$

Multiple-Path Pruning & A^*

- Suppose path p to n was selected, but there is a shorter path to n; and suppose this shorter path is via path p' on the frontier
- Suppose path p' ends at node n'
- $cost(p) + h(n) \leq cost(p') + h(n')$ because p was selected before p'
- $cost(p') + cost(n', n) < cost(p)$ because the path to n via p' is shorter (by assumption)

$$cost(n', n) < cost(p) - cost(p') \leq h(n') - h(n)$$

You can ensure this doesn't occur by letting
$h(n') - h(n) \leq cost(n', n)$

Monotone Restriction

- Heuristic function h satisfies the **monotone restriction** if $h(m) - h(n) \leq cost(m, n)$ for every arc $\langle m, n \rangle$
- **Monotone** heuristic functions are also called **consistent**
- $h(m) - h(n)$ is the heuristic estimate of the path cost from m to n
- The heuristic estimate of the path cost is **always less than** the actual cost
- If h satisfies the monotone restriction, A^* with multiple path pruning **always finds the shortest path** to a goal

Monotonicity and Admissibility

- This is a strengthening of the admissibility criterion
- if $n = g$ so $h(n) = 0$ and $cost(n', n) = cost(n', g) = cost\_to\_goal(n')$, then we can derive from

  $$h(n') \leq cost(n', n) + h(n)$$

  that

  $$h(n') \leq cost\_to\_goal(n')$$

  which is **admissibility**

Monotonicity and Admissibility

- This is a strengthening of the admissibility criterion
- if $n = g$ so $h(n) = 0$ and $cost(n', n) = cost(n', g) = cost\_to\_goal(n')$, then we can derive from

  $$h(n') \leq cost(n', n) + h(n)$$

  that

  $$h(n') \leq cost\_to\_goal(n')$$

  which is **admissibility**
- So Monotonicity is like Admissibility but between **any two nodes**

Summary of Search Strategies

Strategy	Frontier Selection	Halts?	Space	Time
Depth-first	Last node added	No	Linear	Exp
Breadth-first	First node added	Yes	Exp	Exp
Heuristic depth-first	Local[1] min $h(n)$	No	Linear	Exp
Best-first	Global[2] min $h(n)$	No	Exp	Exp
Lowest-cost-first	Minimal $cost(n)$	Yes	Exp	Exp
A^*	Minimal $f(n)$	Yes	Exp	Exp

[1] Locally in some region of the frontier

[2] Globally for all nodes on the frontier

Adversarial Search: Minimax

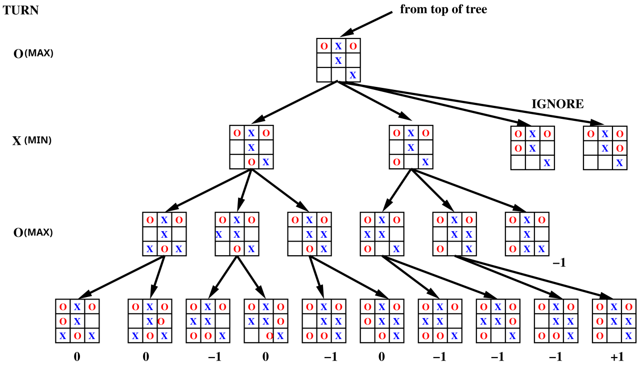
- For **competitive, two-person, zero-sum** games (e.g. tic-tac-toe)
- Try to find the **best option** for you on nodes that you control (called MAX nodes)
- Assume competitor ("X") will take the **worst option** for you on nodes you do not control (called MIN nodes)
- Recursively search to leaf nodes to find **state evaluations**, and percolate values upward through the tree

Minimax Algorithm

function $\text{MINIMAX}(node, depth, isMax)$
  if $depth = 0$ or *node* is a terminal node **then**
    return the heuristic value of *node*
  if *isMax* **then**
    $bestValue \leftarrow -\infty$
    for each *child* of *node* **do**
      $v \leftarrow \text{minimax}(child, depth - 1, \text{False})$
      $bestValue \leftarrow \max(bestValue, v)$
  else
    $bestValue \leftarrow +\infty$
    for each *child* of *node* **do**
      $v \leftarrow \text{minimax}(child, depth - 1, \text{True})$
      $bestValue \leftarrow \min(bestValue, v)$
  return *bestValue*
end function

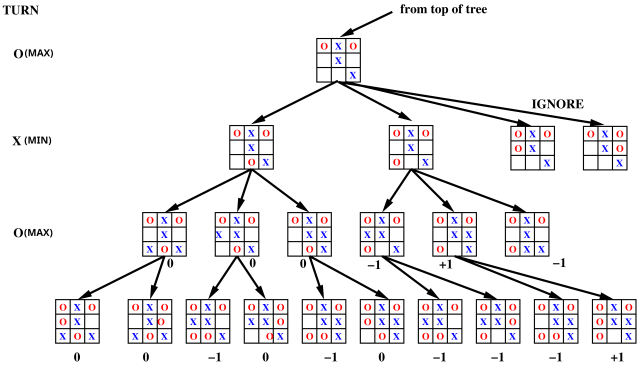
Minimax Example: Tic-Tac-Toe

- We are "O": "O" turns are MAX turns, and "X" turns are MIN turns
- Label each node here with expected reward (-1,0 or +1)
- (Note: some nodes are ignored to save space on the slide)



Minimax Example: Tic-Tac-Toe

- We are "O": "O" turns are MAX turns, and "X" turns are MIN turns
- Label each node here with expected reward (-1,0 or +1)
- (Note: some nodes are ignored to save space on the slide)



Minimax in Larger Games

- Searching all the way to every leaf node is impossibly costly in larger games (e.g. chess)

Minimax in Larger Games

- Searching all the way to every leaf node is impossibly costly in larger games (e.g. chess)
- **Alpha-beta pruning** is a method that allows us to ignore portions of the search tree without losing optimality
  - It is useful in practical application, but does not change worst-case performance (exponential)

Minimax in Larger Games

- Searching all the way to every leaf node is impossibly costly in larger games (e.g. chess)
- **Alpha-beta pruning** is a method that allows us to ignore portions of the search tree without losing optimality
  - It is useful in practical application, but does not change worst-case performance (exponential)
- We can also stop search early by evaluating **non-leaf** nodes via **heuristics**
  - Can no longer guarantee optimal play
  - Can set a fixed maximum depth for the search tree

Higher Level Strategies

The following methods are not full algorithms per se, but can be used to form a strategy for search at a higher level

Direction of Search

- The definition of searching is **symmetric**: find path from start nodes to goal node or from goal node to start nodes
- **Forward branching factor:** number of arcs out of a node
- **Backward branching factor:** number of arcs into a node
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa
- Note: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph

Bidirectional Search

- You can search backward from the goal and forward from the start **simultaneously**
- This wins as $2b^{k/2} \ll b^k$.
  This **can result** in an exponential saving in time and space
- The main problem is making sure the **frontiers meet**
- This is often used with one **breadth-first** method that builds a set of locations that can lead to the goal
  In the other direction another method can be used to find a path to these interesting locations

Island Driven Search

- **Idea:** find a set of **islands** between s and g.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

  There are m **smaller problems** rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$
- The problem is to **identify the islands** that the path must pass through
  It is **difficult to guarantee optimality**

Next

- Constraints (Poole & Mackworth Chap. 4)