Lecture 3 - States and Searching

Jesse Hoey
School of Computer Science
University of Waterloo

May 10, 2022

Readings: Poole & Mackworth Chapt. 3 (all)

Searching

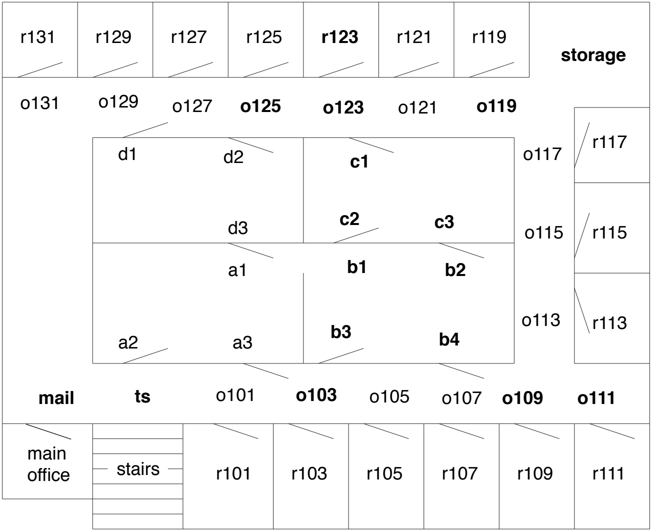
- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.
- A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.
- Many AI problems can be abstracted into the problem of finding a path in a directed graph.
- Often there is more than one way to represent a problem as a graph.

Directed Graphs

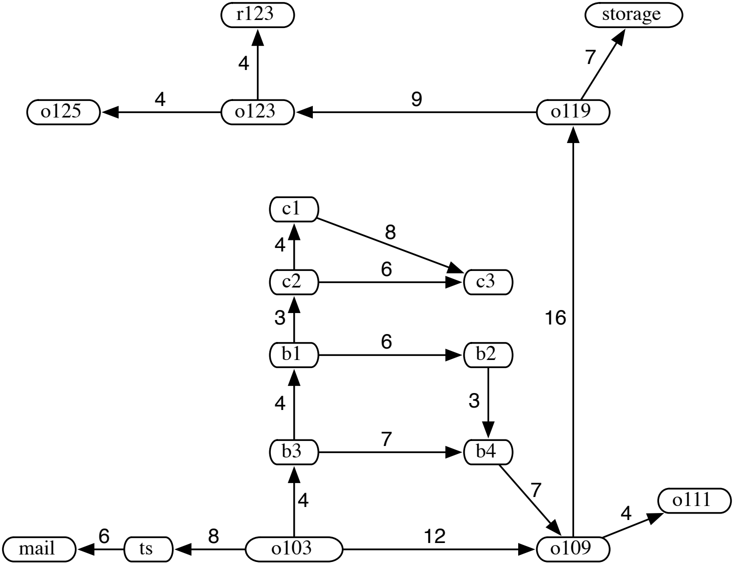
- A graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.
- Often there is a cost associated with arcs and the cost of a path is the sum of the costs of the arcs in the path.

Example Problem for Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.



Graph for the Delivery Robot

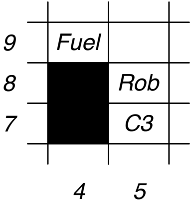


cost = distance travelled

Problem space search

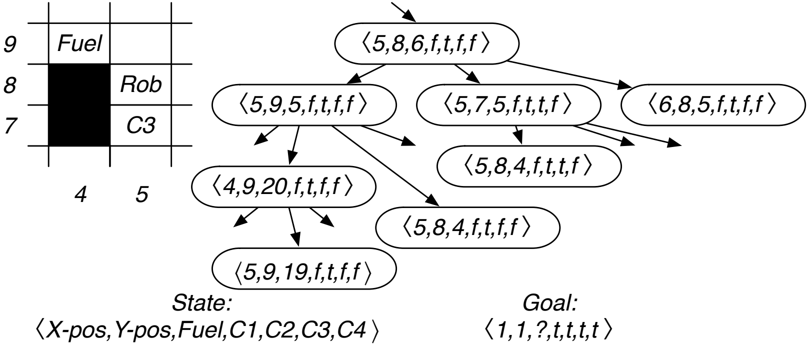
Partial Search Space for a Video Game

Grid game: collect coins C_1, C_2, C_3, C_4, don't run out of fuel, and end up at location $(1, 1)$:



Partial Search Space for a Video Game

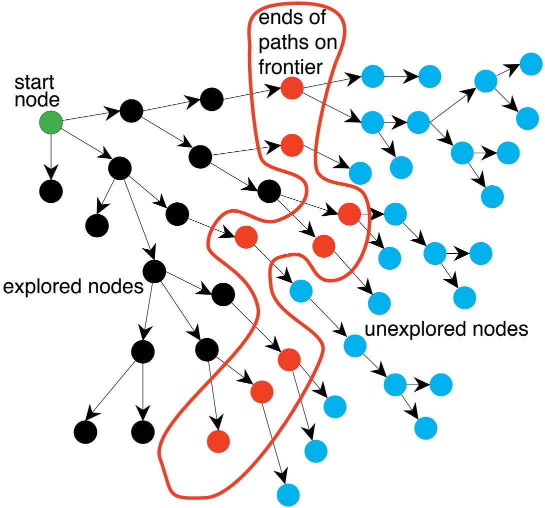
Grid game: collect coins C_1, C_2, C_3, C_4, don't run out of fuel, and end up at location $(1,1)$:



Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

Problem Solving by Graph Searching



Graph Search Algorithm

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
  if $goal(n_k)$
    return $\langle n_0, \ldots, n_k \rangle$;
  for every neighbor n of n_k
    add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while

Graph Search Algorithm

- We assume that after the search algorithm returns an answer, it can be asked for more answers and the procedure continues.
- The neighbors define the graph.
- Which value is selected from the frontier (and how the new values are added to the frontier) at each stage defines the search strategy.
- *goal* defines what is a solution.

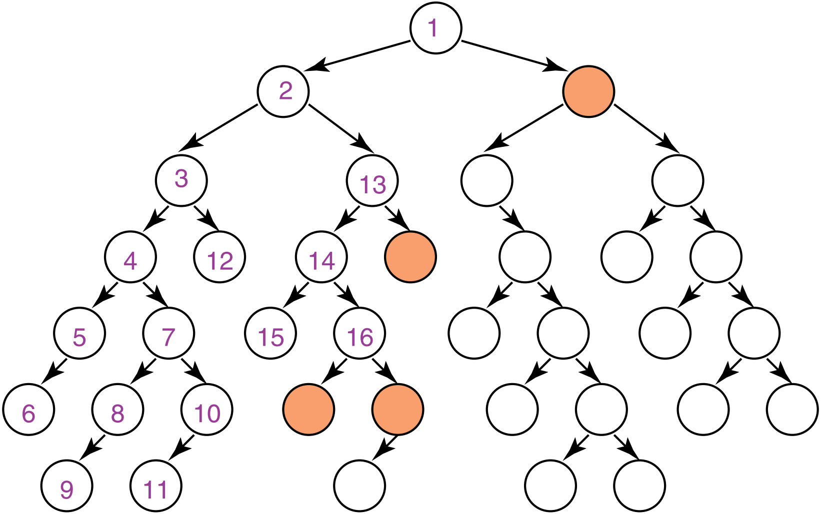
Types of Search

- Uninformed (blind)
- Heuristic
- More sophisticated “hacks”

Depth-first Search

- Depth-first search treats the frontier as a stack
- It always selects the last element added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$
  - p_1 is selected. Paths that extend p_1 are added to the front of the stack (in front of p_2).
  - p_2 is only selected when all paths from p_1 have been explored.

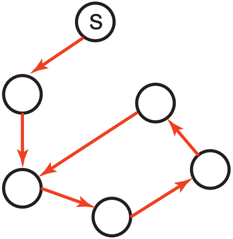
Illustrative Graph — Depth-first Search



Complexity of Depth-first Search

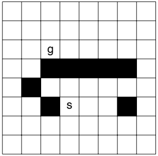
- Depth-first search not guaranteed to halt on infinite graphs or on graphs with cycles.
- The space complexity is linear in the size of the path being explored.
- Search is unconstrained by the goal until it happens to stumble on the goal (uninformed or blind)
- What is the worst-case time complexity of depth-first search?

Cycle Checking



- A searcher can prune a path that ends in a node already on the path,
- Using depth-first methods, with the graph explicitly stored, this can be done in constant time (add a flag to each node)
- For other methods, the cost is linear in path length, since we only have to check for cycles in the current path.

Graph Search Algorithm - with Cycle Check



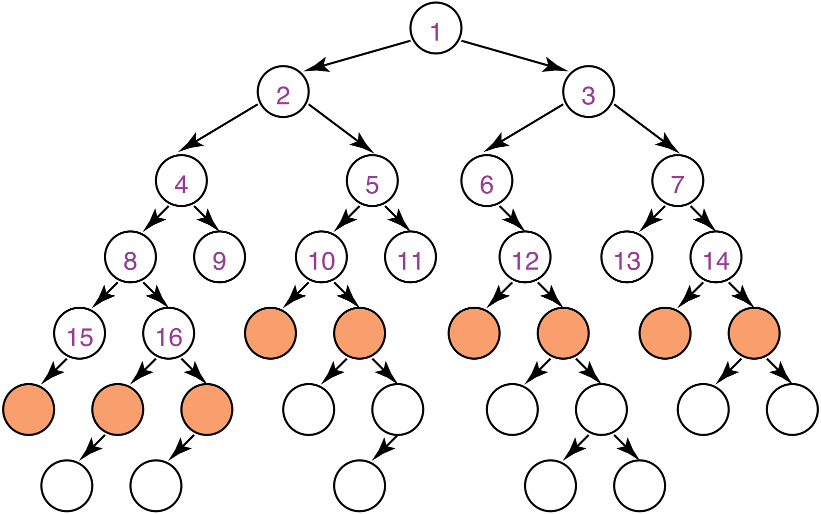
- Use Depth First Search to get from **s** to **g**
- Number the nodes as they are removed
- add neighbours CCW from top L,D,R,U
- Use a cycle check

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
frontier $:= \{\langle s \rangle : s$ is a start node$\}$;
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
  if $goal(n_k)$
    return $\langle n_0, \ldots, n_k \rangle$;
  for every neighbor n of n_k
    if $n \notin \langle n_0, \ldots, n_k \rangle$
      add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while

Breadth-first Search

- Breadth-first search treats the frontier as a queue.
- It always selects hte earliest element added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$:
  - p_1 is selected. Its neighbors are added to the end of the queue, after p_r.
  - p_2 is selected next.

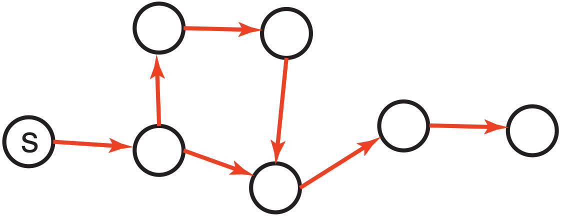
Illustrative Graph — Breadth-first Search



Complexity of Breadth-first Search

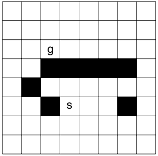
- The branching factor of a node is the number of its neighbors.
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists.
  It is guaranteed to find the path with fewest arcs.
- Time complexity is exponential in the path length: b^n, where b is branching factor, n is path length.
- Space complexity is exponential in path length: b^n.
- Search is unconstrained by the goal.
- Not affected by cycles (remains exponential).

Multiple-Path Pruning



- Multiple path pruning: prune a path to node n that any path has been found to.
- Multiple-path pruning subsumes a cycle check (because the current path is a path to the node).
- This entails storing all nodes it has found paths to.
- Want to guarantee that an optimal solution can still be found. (See slide 40)

Graph Search Algorithm - with Multiple Path Pruning



- Use Breadth First Search to get from **s** to **g**
- Number the nodes as they are removed
- add neighbours CW from top U,R,D,L
- Use multiple path pruning

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
$has\_path := \{\}$;
while $frontier$ is not empty:
  select and **remove** path $\langle n_0, \dots, n_k \rangle$ from $frontier$;
  if $n_k \notin has\_path$:
    add n_k to has_path;
    if $goal(n_k)$
      return $\langle n_0, \dots, n_k \rangle$;
    for every neighbor n of n_k
      add $\langle n_0, \dots, n_k, n \rangle$ to $frontier$;
end while

Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} |\langle n_{i-1}, n_i \rangle|$$

- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- It finds a least-cost path to a goal node
- When arc costs are equal $\Longrightarrow$ breadth-first search.
- Uniformed/Blind search (in that it does not take the goal into account)
- Complexity: exponential

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 $h(n)$.

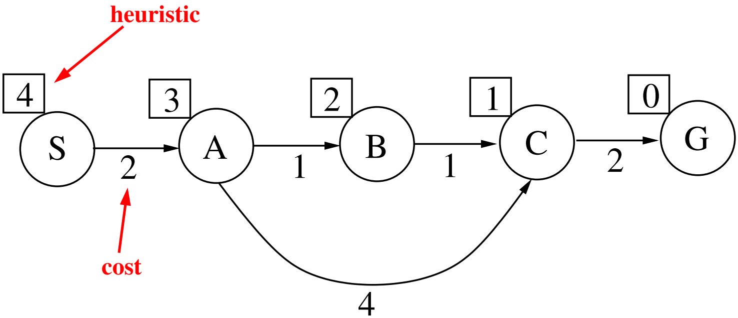
Example Heuristic Functions

- 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 $h(n)$.
- 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

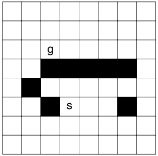
- Idea: select the path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal h-value.
- It treats the frontier as a priority queue ordered by h.

Illustrative Example — Best First Search



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

Graph Search Algorithm - with Multiple Path Pruning



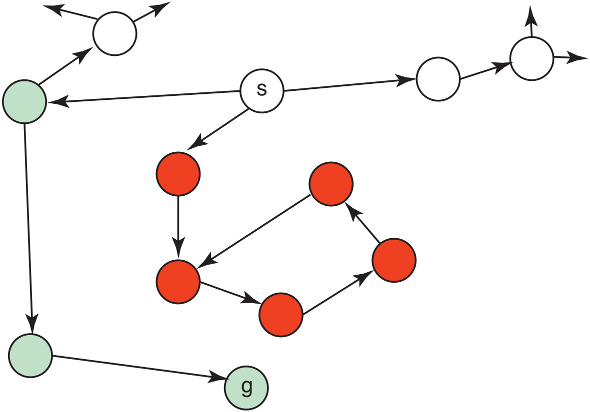
- Use Best First Search to get from **s** to **g**
- Number the nodes as they are removed
- Use multiple path pruning
- break ties arbitrarily
- Use Manhattan Distance as heuristic

Input: a graph + start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
$has\_path := \{\}$;
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
  if $n_k \notin has\_path$:
    add n_k to has_path;
    if $goal(n_k)$
      return $\langle n_0, \ldots, n_k \rangle$;
    for every neighbor n of n_k
      add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while

Heuristic Depth-first Search

- Idea: Do a depth-first seach, but add paths to the stack ordered according to h
- Locally does a best-first search, but agressively pursues the best looking path (even if it ends up being worse than one higher up).
- Suffers from the same problems as depth-first search
- Is often used in practice

Illustrative Graph — Heuristic Search



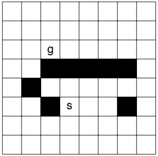
cost of an arc is its length

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: a graph + start nodes
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
$has\_path := \{\}$;
while $frontier$ is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from $frontier$;
  if $n_k \notin has\_path$:
    add n_k to has_path;
    if $goal(n_k)$
      return $\langle n_0, \ldots, n_k \rangle$;
    for every neighbor n of n_k
      add $\langle n_0, \ldots, n_k, n \rangle$ to $frontier$;
end while

A^* Search

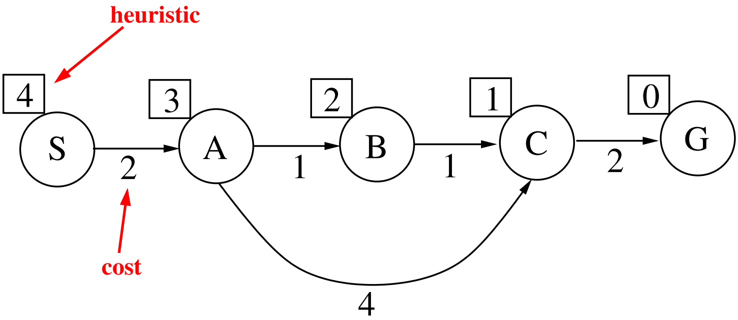
- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$. $f(p)$ estimates the total path cost of going from a start node to a goal via p.

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

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^* sxearch



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

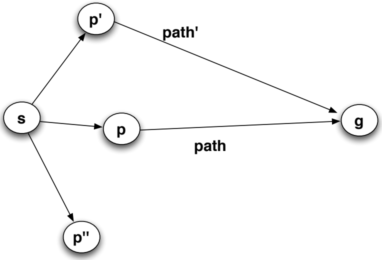
Admissibility of A^*

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), and
- $h(n)$ is a lower bound on the length (cost) of the shortest path from n to a goal node.

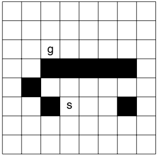
Admissible heuristics never overestimate the cost to the goal.

Why is A^* with admissible h optimal?



- assume: $paths \rightarrow p \rightarrow g$ 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

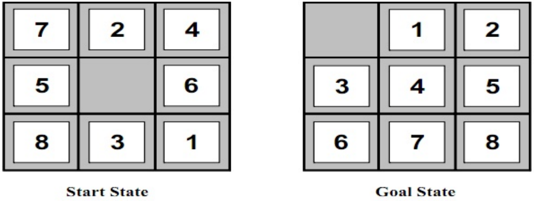


- Use A* search
- Number the nodes as they are removed
- Use multiple path pruning
- break ties arbitrarily
- Use Manhattan Distance as heuristic

Input: a graph and set of start nodes,
Boolean procedure *goal*(n) that tests if n is a goal node.
frontier $:= \{\langle s\rangle : s$ is a start node$\}$;
has_path $:= \{\}$;
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k\rangle$ from *frontier*;
  if $n_k \notin$ *has_path*:
    add n_k to *has_path*;
    if *goal*(n_k)
      return $\langle n_0, \ldots, n_k\rangle$;
    for every neighbor n of n_k
      add $\langle n_0, \ldots, n_k, n\rangle$ to *frontier*;
end while

How do we construct a heuristic?

"magic square" tiles can move into adjacent empty slot only

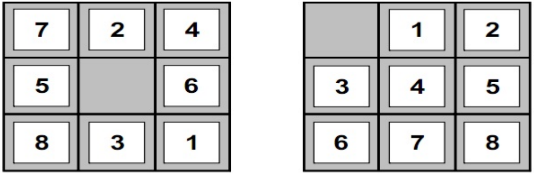


Relax the game (make it simpler, easier)

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

How do we construct a heuristic?

"magic square" tiles can move into adjacent empty slot only



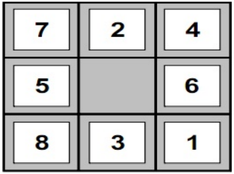
Start State **Goal State**

Relax the game (make it simpler, easier)

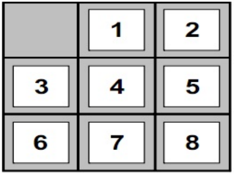
1. Can move tile from position A to position B if A is next to B (ignore whether or not position is blank)
   - leads to manhattan distance heuristic
   - To solve the puzzle need to slide each tile into its final position
   - Admissible

How do we construct a heuristic?

"magic square" tiles can move into adjacent empty slot only



Start State



Goal State

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

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

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. 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 if $h(n') - h(n) \leq cost(n', n)$.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if $h(m) - h(n) \leq cost(m, n)$ for every arc $\langle m, n \rangle$.
- $h(m) - h(n)$ is the heuristic estimate of the path cost from m to n
- The heurstic 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 $cost(n', n) = cost(n')$, then we can derive from

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

  that

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

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

Iterative Deepening

- So far all search strategies that are guaranteed to halt use exponential space.
- Idea: let's recompute elements of the frontier rather than saving them.
- Look for paths of depth 0, then 1, then 2, then 3, etc.
- You need a depth-bounded depth-first searcher.
- If a path cannot be found at depth B, look for a path at depth $B + 1$. Increase the depth-bound when the search fails unnaturally (depth-bound was reached).

Iterative Deepening Complexity

Complexity with solution at depth k & branching factor b:

	# times each node is expanded		
level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
…	…	…	…
$k-1$	1	2	b^{k-1}
k	1	1	b^k
	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

$$b^k + 2b^{k-1} + 3b^{k-2} + \ldots = b^k \sum_{n=1}^{k} n \left(\frac{1}{b}\right)^{n-1} \quad \text{rewrite} \tag{1}$$

$$< b^k \sum_{n=1}^{\infty} n \left(\frac{1}{b}\right)^{n-1} \quad \text{extend to infinity} \tag{2}$$

$$= b^k \left(\frac{b}{1-b}\right)^2 \quad \text{derivative of the geometric series} \tag{3}$$

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.

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

  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.
- You can solve the subproblems using islands $\Longrightarrow$ hierarchy of abstractions.

Dynamic Programming

- Start from goal and work backwards
- Compute the cost-to-goal at each node recursively
- e.g. Dijkstra's algorithm
- Cost from $n \rightarrow$ goal is
  Cost from $m \rightarrow$ goal + cost from n to m
- $dist(n)$ is cost-to-goal from node n, and $cost(n, m)$ is cost to go from n to m

$$dist(n) = \begin{cases} 0 & \text{if } n \text{ is goal} \\ \min_m(cost(n, m) + dist(m)) & \text{otherwise} \end{cases}$$

- $dist(n)$ is a value function over nodes
- policy(n) is best m for each n, so best path is

$$path(n, goal) = \arg\min_m(cost(n, m) + dist(m))$$

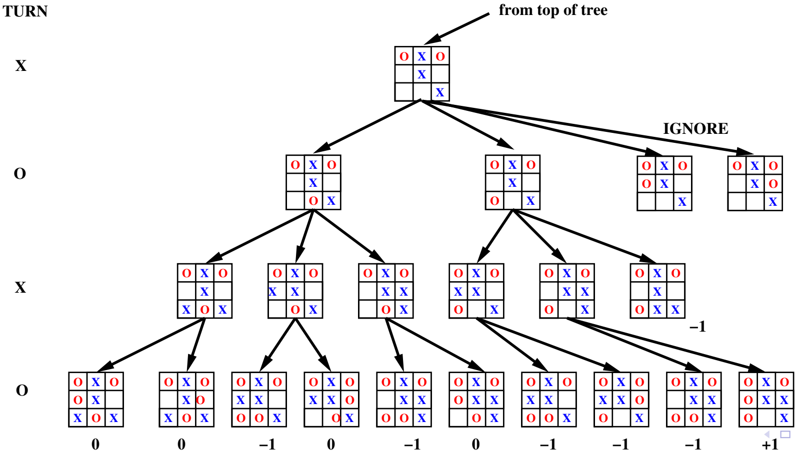
- problem: space needed to store entire graph

Discounted Dynamic Programming

- assume goal has a reward, $R(n)$
- arcs still have costs
- Rewards far in the future are less valuable
- "a bird in the hand is worth two in the bush"
- Discount factor $\beta < 1$
- maximize rewards
- $$dist(n) = \begin{cases} R(n) & \text{if } n \text{ is goal} \\ \max_m(\beta dist(m) - cost(n,m)) & \text{otherwise} \end{cases}$$

Minimax Search

- for competitive, two-person, zero-sum games (tic-tac-toe)
- try to find the best option for you ("O")
- assume competitor ("X") will take the worst option for you
- label each node here with expected reward (-1,0 or +1)



Next:

- Constraints (Poole & Mackworth chapter 4)