Chapter 3: Search

Learning Objectives

At the end of the class you should be able to:

- define a directed graph
- represent a problem as a state-space graph
- explain how a generic searching algorithm works

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.

State-space Search

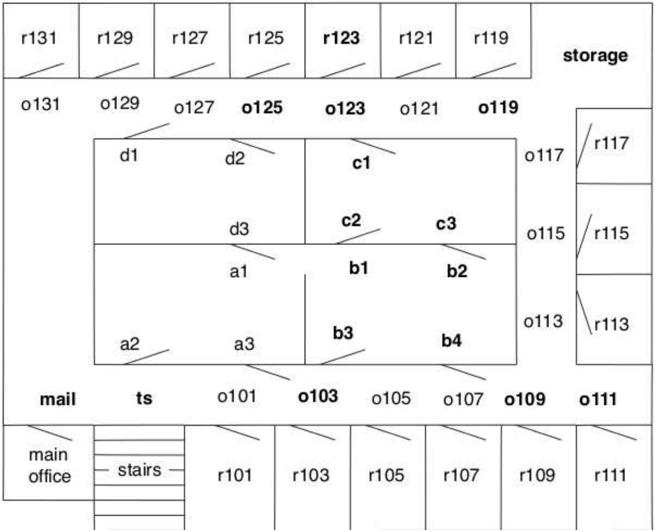
- flat or modular or hierarchical
- explicit states or features or individuals and relations
- static or finite stage or indefinite stage or infinite stage
- fully observable or partially observable
- deterministic or stochastic dynamics
- goals or complex preferences
- single agent or multiple agents
- knowledge is given or knowledge is learned
- perfect rationality or bounded rationality

Directed Graphs

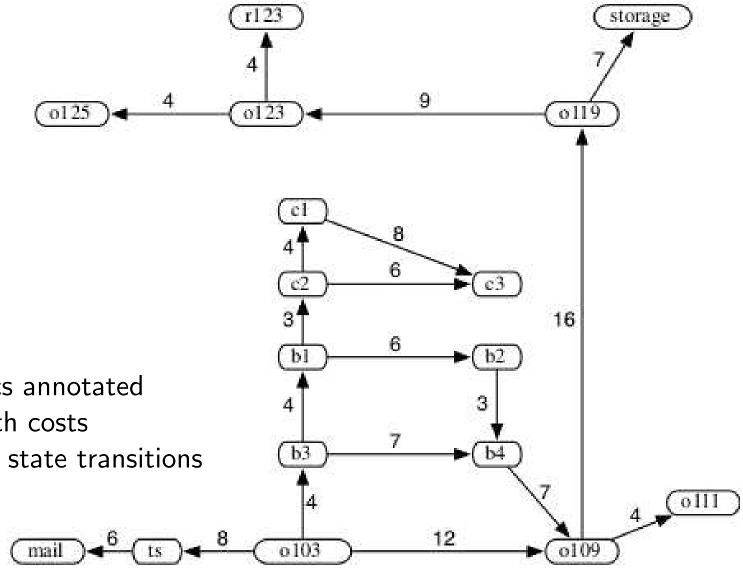
- A graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1},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

Example Problem for Delivery Robot

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



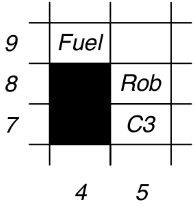
State-Space Graph for the Delivery Robot



arcs annotated
with costs
for state transitions

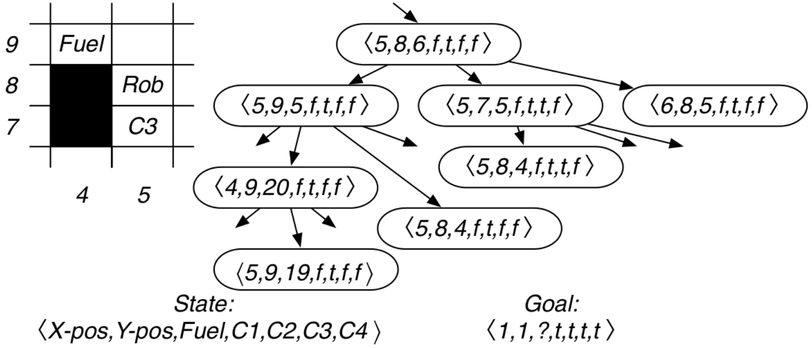
Partial Search Space for a Video Game

Grid game: Rob needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1, 1)$:



Partial Search Space for a Video Game

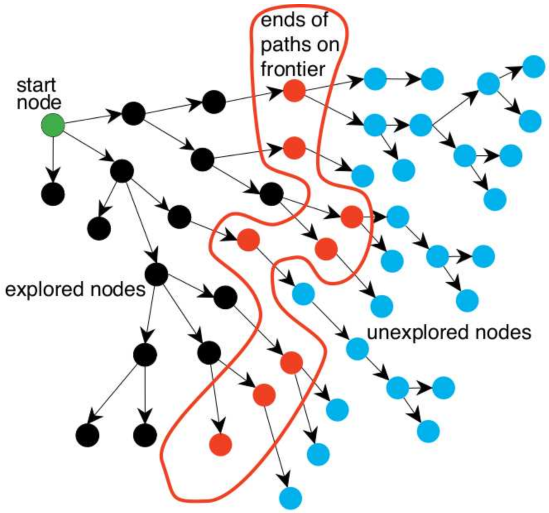
Grid game: Rob needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location $(1,1)$:



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.
$frontier := \{\langle s \rangle : s$ is a start node$\}$;
while $frontier$ is not empty:
       select and remove path $\langle n_0, \dots, n_k \rangle$ from $frontier$;
       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
```

- Which value is selected from the frontier at each stage defines the search strategy.
- The neighbors define the graph.
- *goal* defines what is a solution.
- If more than one answer is required, the search can continue from the return.

Learning Objectives

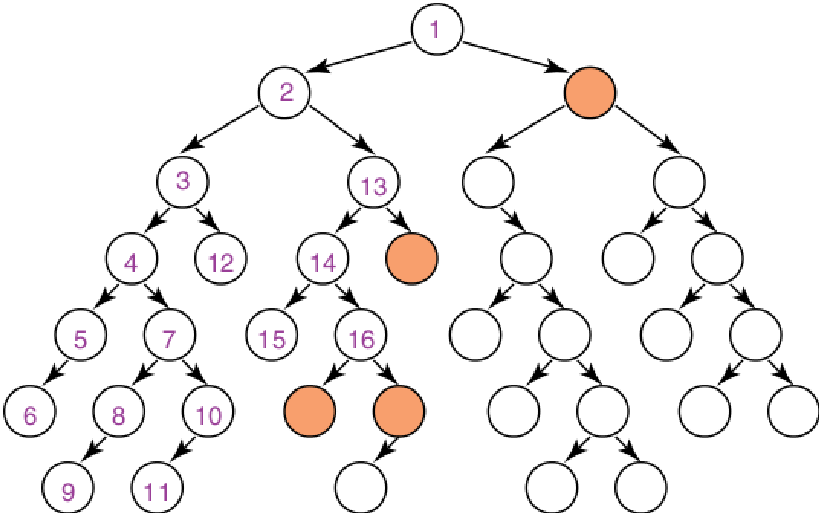
At the end of the class you should be able to:

- demonstrate how depth-first search will work on a graph
- demonstrate how breadth-first search will work on a graph
- predict the space and time requirements for depth-first and breadth-first searches

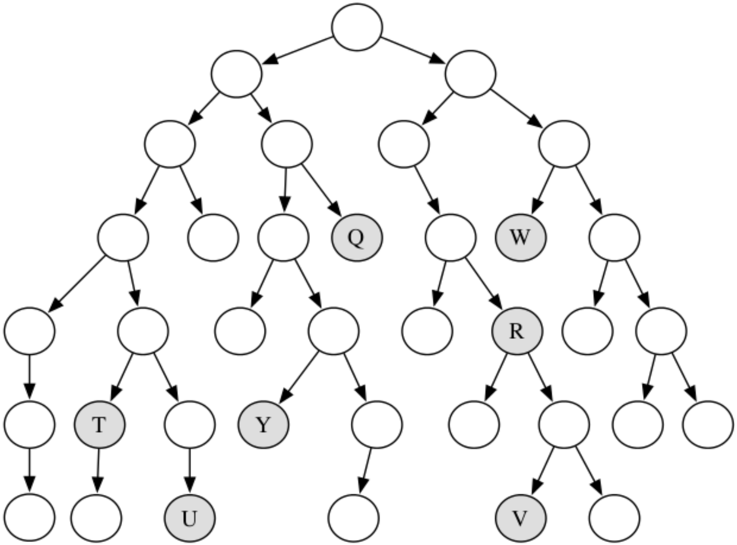
Depth-first Search

- Depth-first search treats the frontier as a stack
- It always selects one of the last elements 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



Which shaded goal will a depth-first search find first?



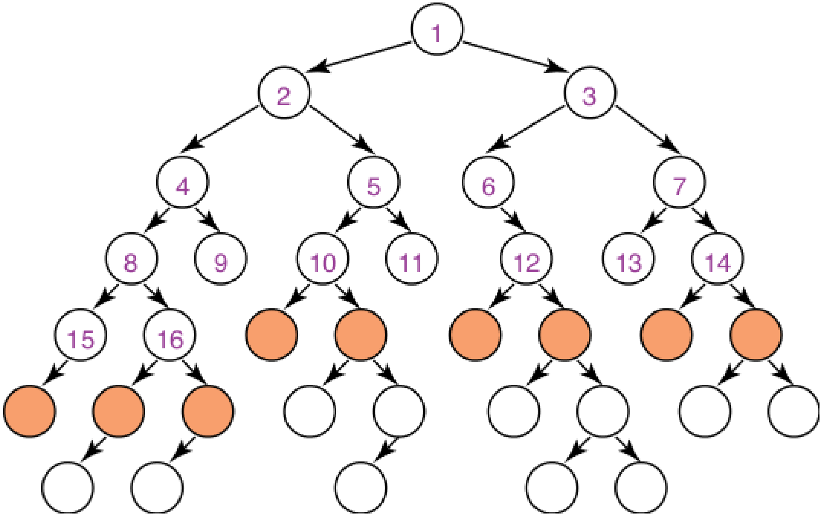
Properties of Depth-first Search

- Does depth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Breadth-first Search

- Breadth-first search treats the frontier as a queue.
- It always selects one of the earliest elements 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.

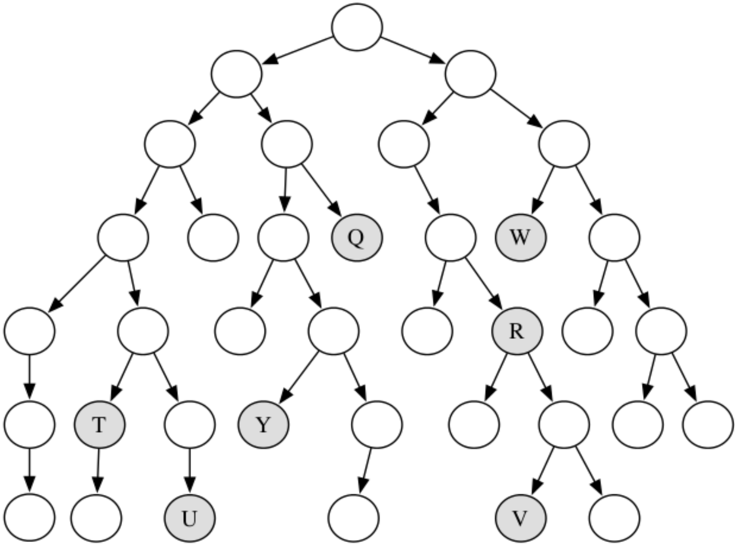
Illustrative Graph — Breadth-first Search



Properties of Breadth-first Search

- Does breadth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?
- What is the space complexity as a function of the length of the path selected?
- How does the goal affect the search?

Which shaded goal will a breadth-first search find first?



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.

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

  An optimal solution is one with minimum cost.
- 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.

Properties of Heuristic Depth-first Search

- Does heuristic depth-first search guarantee to find the path with the lowest cost?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?
- What is the space complexity as a function of the length of the path selected?
- How does the goal affect the search?

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Learning Objectives

At the end of the class you should be able to:

- devise a useful heuristic function for a problem
- demonstrate how best-first and A^* search will work on a graph
- predict the space and time requirements for best-first and A^* search

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.
- $h(n)$ needs to be efficient to compute.
- 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 with cost less than $h(n)$.
- An admissible heuristic is a nonnegative heuristic function that is an underestimate of the actual cost of a path to a goal.

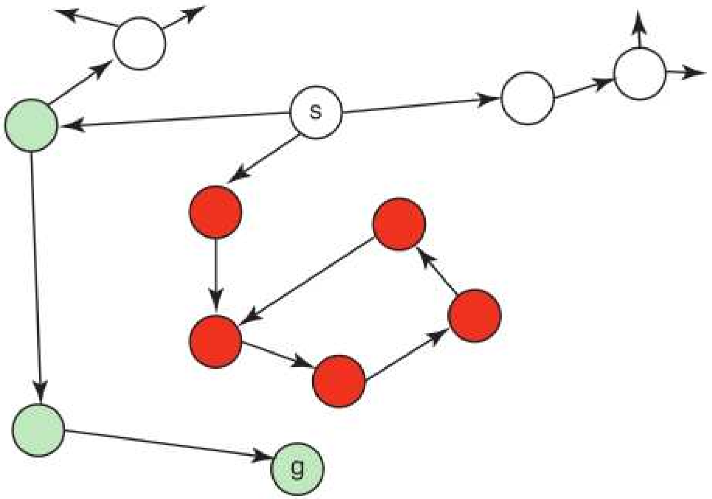
Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel, the cost is an estimate of how many steps it will take to collect the rest of the coins, refuel when necessary, and return to goal position.
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

Heuristic Depth-first Search

- It's a way to use heuristic knowledge in depth-first search.
- Idea: order the neighbors of a node (by their *h*-value) before adding them to the front of the frontier.
- It locally selects which subtree to develop, but still does depth-first search. It explores all paths from the node at the head of the frontier before exploring paths from the next node.

Illustrative Graph — Heuristic Depth-first Search



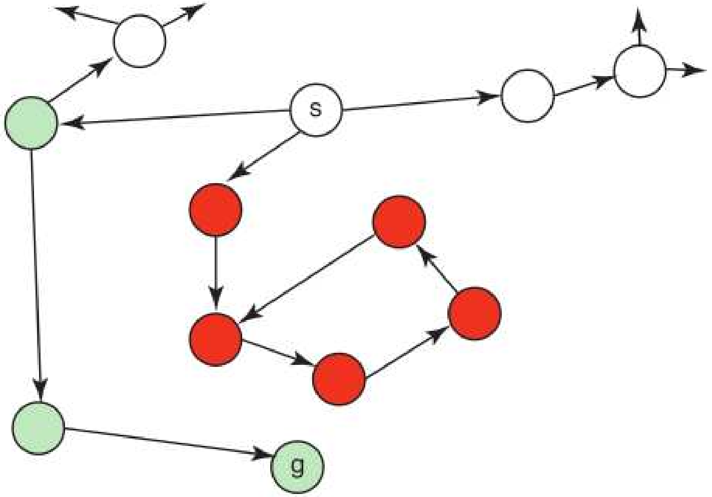
Properties of Heuristic Depth-first Search

- Does heuristic depth-first search guarantee to find the shortest path or the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

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 Graph — Best-first Search



Properties of Best-first Search

- Does best-first search guarantee to find the shortest path or the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

A^* Search

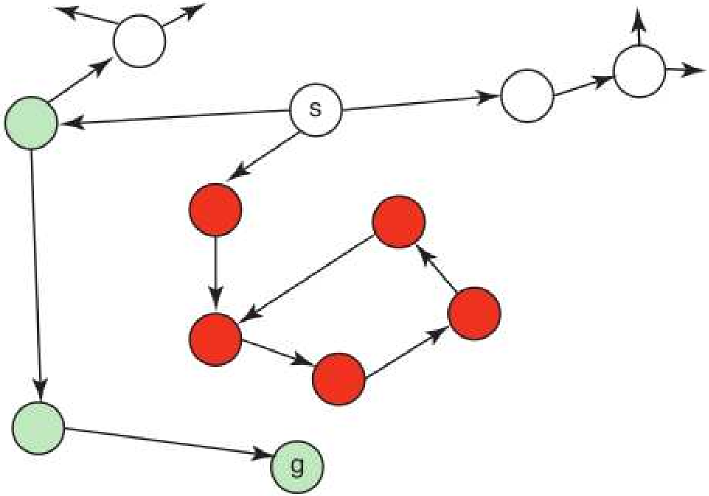
- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$.
  $f(p)$ estimates the total path cost of going from a start node to a goal via p.

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

A^* Search Algorithm

- A^* is a combination 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 Graph — A^* Search



Properties of A^* Search

- Does A^* search guarantee to find the shortest path or the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

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 nonnegative and an underestimate of the cost of the shortest path from n to a goal node.

Why is A^* admissible?

- If a path p to a goal is selected from a frontier, can there be a shorter path to a goal?
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because h is an underestimate:

$$cost(p') + h(p') \leq cost(p'')$$

  for any path p'' to a goal that extends p'.
- So $cost(p) \leq cost(p'')$ for any other path p'' to a goal.

Why is A^* admissible?

A^* can always find a solution if there is one:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.
- A^* halts, as the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happen to a path p where

- $cost(p) + h(p) < c$
- $cost(p) + h(p) = c$
- $cost(p) + h(p) > c$

How can a better heuristic function help?

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp
Heuristic depth-first	Local min $h(p)$	No	No	Linear
Best-first	Global min $h(p)$	No	No	Exp
A^*	Minimal $f(p)$	Yes	No	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Summary of Search Strategies

Relevant criteria for comparison:

- finite graph / infinite graph
- cycle free / with cycles
- finding one solution / finding all solutions
- solution exists / no solution available

Learning Objectives

At the end of the class you should be able to:

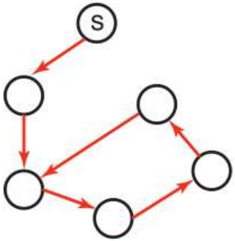
- explain how cycle checking and multiple-path pruning can improve efficiency of search algorithms
- explain the complexity of cycle checking and multiple-path pruning for different search algorithms
- justify why the monotone restriction is useful for A^* search
- predict whether forward, backward, bidirectional or island-driven search is better for a particular problem
- demonstrate how dynamic programming works for a particular problem

Summary of Search Strategies

Relevant criteria for comparison:

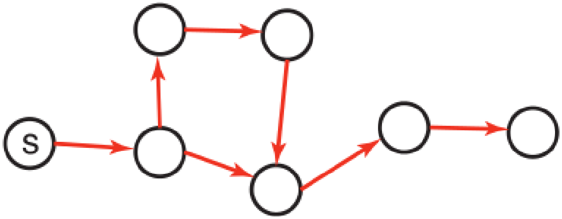
- finite graph / infinite graph
- cycle-free / with cycles
- finding one solution / finding all solutions
- solution exists / no solution available

Cycle Checking



- A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.
- In depth-first methods, checking for cycles can be done in ____________ time in path length.
- For other methods, checking for cycles can be done in ____________ time in path length.
- Does cycle checking mean the algorithms halt on finite graphs?

Multiple-Path Pruning



- Multiple path pruning: prune a path to node *n* that the searcher has already found a path to.
- What needs to be stored?
- How does multiple-path pruning compare to cycle checking?
- Do search algorithms with multiple-path pruning always halt on finite graphs?
- What is the space & time overhead of multiple-path pruning?
- Can multiple-path pruning prevent an optimal solution being found?

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 or
- change the initial segment of the paths on the frontier to use the shorter path or
- ensure this doesn't happen. Make sure that the shortest path to a node is found first.

Multiple-Path Pruning & A^*

- Can we make sure that the shortest path to a node is always found first?
- 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'.
- p was selected before p', so:
  $cost(p) + h(n) \leq cost(p') + h(n')$.
- Suppose $cost(n', n)$ is the actual cost of a path from n' to n. The path to n via p' is shorter than p so:
  $cost(p') + cost(n', n) < cost(p)$.

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

We 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 $\langle m, n \rangle$.
- If h satisfies the monotone restriction, A^* with multiple path pruning always finds the shortest path to a goal.
- This is a strengthening of the admissibility criterion.

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.
- A depth-bounded depth-first searcher can do this in linear space.
- 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 search

```
Boolean natural_failure;
Procedure $dbsearch(\langle n_0, \ldots, n_k \rangle : path, bound : int)$:
    if $goal(n_k)$ and $bound = 0$ report path $\langle n_0, \ldots, n_k \rangle$;
    if $bound > 0$
        for each neighbor $n$ of $n_k$
                $dbsearch(\langle n_0, \ldots, n_k, n \rangle, bound - 1)$;
    else if $n_k$ has a neighbor then $natural\_failure := false$;
end procedure $dbsearch$;
Procedure $idsearch(S : node)$:
    Integer $bound := 0$;
    repeat
        $natural\_failure := true$;
        $dbsearch(\langle s \rangle, bound)$;
        $bound := bound + 1$;
    until $natural\_failure$;
end procedure $idsearch$
```

Complexity of Iterative Deepening

Complexity with solution at depth k & branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Depth-first Branch-and-Bound

- Way to combine depth-first search with heuristic information.
- Finds optimal solution.
- Most useful when there are multiple solutions, and we want an optimal one.
- Uses the space of depth-first search.

Depth-first Branch-and-Bound

- Idea: maintain the cost of the lowest-cost path found to a goal so far, call this *bound*.
- If the search encounters a path p such that $cost(p) + h(p) \geq bound$, path p can be pruned.
- If a non-pruned path to a goal is found, it must be better than the previous best path. This new solution is remembered and *bound* is set to its cost.
- The search can be a depth-first search to save space.
- How should the bound be initialized?

Depth-first Branch-and-Bound: Initializing Bound

- The bound can be initialized to ∞.
- The bound can be set to an estimate of the optimal path cost.
  - if the bound is slightly larger than the cost of the optimal path, branch-and-bound does not expand more nodes than A^*
- idea can be used to iteratively approach the optimal solution (similar to iterative deepening)

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: when graph is dynamically constructed, the backwards graph may not be available.

Bidirectional Search

- Idea: 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.
- The subproblems can be solved using islands $\Longrightarrow$ hierarchy of abstractions.

Dynamic Programming

Idea: A partial solution path up to a state will be part of the globally optimal solution, if the state lies on the globally optimal solution path. (BELLMAN 1957)

Solution: for statically stored graphs, build a table of $cost(n)$ to reach (or leave) a node.

$cost(n)$ is computed recursively:

$$cost(n) = \min_{\forall m. n = succ(m)} cost(m) + cost(\langle m, n \rangle)$$

Dynamic Programming

Dynamic programming can be applied, if

- an optimal path has to be found,
- the goal state is known in advance, and
- the graph is small enough to maintain the complete distance table for a given goal.

Dynamic Programming

Examples:

- string-to-string mapping (spelling correction, gene sequence analysis, speech recognition, ...)
- structural classification with probabilistic models (tagging, parsing, translation, composite object recognition, ...)

Efficient solutions, if

- the branching factor is quite small,
- the branching of paths is balanced by their recombination
- node identity can be checked in constant time

String-to-String Mapping

Stepwise alignment of the two strings considering four different cases:

- identity: the current characters in both strings are the same
- substitution: the current character in string A has been replaced by another one in string B
- deletion: the current character in string A does not exist in string B
- insertion: the current character in string B does not exist in string A

String-to-String Mapping

How to recast string mapping as a search problem?

- states, state descriptions
- start / goal state
- state transitions
- branching factor
- size of the graph

String-to-String Mapping

Simplest cost model (LEVENSTEIN-metric):

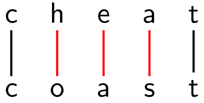
$$cost(id) = 0$$
$$cost(sub) = cost(del) = cost(ins) = 1$$

More sophisticated cost functions can capture additional domain knowledge

- neighbourhood on a keyboard
- phonetic similarities
- user specific confusions
- ...

String-to-String Mapping

Alternative alignments with the same distance are possible



String-to-String Mapping

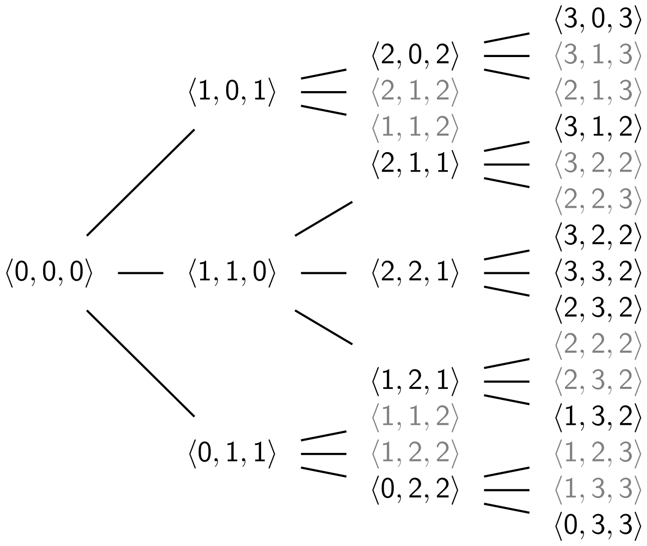
- Representation of search states

$$\langle \mathit{position\_in\_A}, \mathit{position\_in\_B}, \mathit{costs} \rangle$$

- State transitions

$$\langle i, j, c_{old} \rangle \Rightarrow \begin{cases} \langle i+1, j+1, c_{new} \rangle & c_{new} = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{else} \end{cases} \\ \langle i+1, j, c_{old}+1 \rangle & \\ \langle i, j+1, c_{old}+1 \rangle & \end{cases}$$

String-to-String Mapping



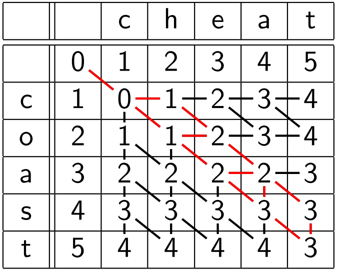
String-to-String Mapping

Populating the distance table

local distances

		c	h	e	a	t
	0	1	1	1	1	1
c	1	0	1	1	1	1
o	1	1	1	1	1	1
a	1	1	1	1	0	1
s	1	1	1	1	1	1
t	1	1	1	1	1	0

global distances



		c	h	e	a	t
	0	1	2	3	4	5
c	1	0	1	2	3	4
o	2	1	1	2	3	4
a	3	2	2	2	2	3
s	4	3	3	3	3	3
t	5	4	4	4	4	3

Isolated Word Recognition

- speech is described as a sequence of feature vectors
- recognizer maintains a list of candidate words
- task: find the word among the candidates with the highest similarity to the recognition target.
- the (global) similarity between two words can be accumulated from the (local) similarity of pairs of feature vectors
- the similarity of two feature vectors can be computed as the inverse of the dissimilarity/distance between them, e.g. Euclidean distance

$$dist(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Isolated Word Recognition

Dynamic time warping

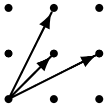
- the same word spoken by the same person varies considerably in its temporal characteristics
- the degree of temporal variation changes over time
- task: find the *optimal alignment* between a candidate word and the recognition target which maximizes global similarity

Constraining the degree of temporal variation: only single feature vectors can be skipped: slope constraint

Isolated Word Recognition

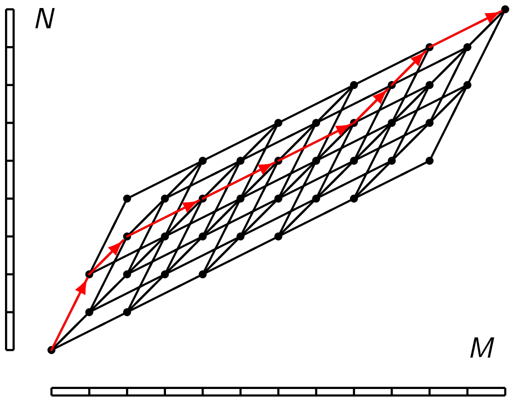
Symmetric slope constraint (SAKOE-CHIBA with deletions)

$$succ(s_{m,n}) = \begin{cases} s_{m+1,n+1} \\ s_{m+2,n+1} \\ s_{m+1,n+2} \end{cases}$$



Isolated Word Recognition

- search graph



Isolated Word Recognition

- First success story of speech recognition
- Search can be implemented in a time-synchroneous manner

- Highly speaker dependent solutions
- Limited vocabulary
- No continuous speech recognition

- Possible application areas:
  - command recognition
  - (numerical) data entry

Dynamic Programming

Application of Dynamic Programming to goal directed search:

Build a table of $dist(n)$ the actual distance of the shortest path from node n to a goal.

The table can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is\_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

$dist(n)$ is an optimal policy to reach the goal from state n.

Knowing $dist(n)$, the choice of the optimal path is a deterministic one.

Dynamic Programming

Dynamic programming is particularly useful, if

- the problem space ist stable,
- the goal does not change very often, and
- the policy can be reused several times.

Main problems:

- Time and space requirements are linear in the size of the search graph, but graph size is often exponential in the path length.
- Computing the minimum remaining cost can only be done in a breadth-first manner: Enough space is needed to store the graph.
- The *dist* function needs to be recomputed for each goal.