Uninformed Search

CPSC 322 Lecture 5

January 13, 2006
Textbook §2.4
Lecture Overview

Recap

Searching

Depth-First Search

Breadth-First Search

Search with Costs
What we want to be able to do:

- find a solution when we are not given an algorithm to solve a problem, but only a specification of what a solution looks like
- idea: search for a solution

What we need:

- A set of states
- A start state
- A goal state or set of goal states
  - or, equivalently, a goal test: a boolean function which tells us whether a given state is a goal state
- A set of actions
- An action function: a mapping from a state and an action to a new state
A graph consists of
- a set $N$ of nodes;
- a set $A$ of ordered pairs of nodes, called arcs or edges.

Node $n_2$ is a neighbor of $n_1$ if there is an arc from $n_1$ to $n_2$.
- i.e., 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 start node and a set of goal nodes, a solution is a path from the start node to a goal node.
Uninformed Search

Problem Solving by Graph Searching

- Start node
- Frontier
- Explored nodes
- Unexplored nodes
Graph Search Algorithm

**Input:** a graph, a set of start nodes, Boolean procedure $\text{goal}(n)$ that tests if $n$ is a goal node.

$\text{frontier} := \{s : s \text{ is a start node}\}$;

while $\text{frontier}$ is not empty:
  select and remove path $\langle n_0, \ldots, n_k \rangle$ from $\text{frontier}$;
  if $\text{goal}(n_k)$
    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 $\text{frontier}$;

end while

- After the algorithm returns, it can be asked for more answers and the procedure continues.
- Which value is selected from the frontier defines the search strategy.
- The $\text{neighbor}$ relationship defines the graph.
- The $\text{goal}$ function defines what is a solution.
- The **forward branching factor** of a node is the number of arcs going out of that node.
- The **backward branching factor** of a node is the number of arcs going into the node.
- If the forward branching factor of every node is $b$ and the graph is a tree, how many nodes are exactly $n$ steps away from the start node?
  - $b^n$ nodes.
- We’ll assume that all branching factors are finite.
Comparing Algorithms

- **Completeness**
  - if at least one solution exists, the algorithm is guaranteed to find a solution within a finite amount of time

- **Time Complexity**
  - in terms of the maximum path length $m$, and the maximum branching factor $b$, what is the worst-case amount of time that the algorithm will take to run?

- **Space Complexity**
  - in terms of $m$ and $b$, what is the worst-case amount of memory that the algorithm must use?
Depth-first Search

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

**Example:**
- the frontier is \([p_1, p_2, \ldots, p_r]\)
- neighbours of \(p_1\) are \(\{n_1, \ldots, n_k\}\)

**What happens?**
- \(p_1\) is selected, and tested for being a goal.
- Neighbours of \(p_1\) replace \(p_1\) at the beginning of the frontier.
- Thus, the frontier is now \([n_1, \ldots, n_k, p_2, \ldots, p_r]\).
- \(p_2\) is only selected when all paths from \(p_1\) have been explored.
Illustrative Graph — Depth-first Search
Analysis of Depth-first Search

- Is DFS complete?
  - Depth-first search isn't guaranteed to halt on infinite graphs or on graphs with cycles.
  - However, DFS is complete for finite trees.
- What is the time complexity, if the maximum path length is $m$ and the maximum branching factor is $b$?
  - The time complexity is $O(b^m)$: must examine every node in the tree.
  - Search is unconstrained by the goal until it happens to stumble on the goal.
- What is the space complexity?
  - Space complexity is $O(bm)$: the longest possible path is $m$, and for every node in that path must maintain a fringe of size $b$. 
Using Depth-First Search

- When is DFS **appropriate**?
  - space is restricted
  - solutions tend to occur at the same depth in the tree
  - you know how to order nodes in the list of neighbours so that solutions will be found relatively quickly

- When is DFS **inappropriate**?
  - some paths have infinite length
  - the graph contains cycles
  - some solutions are very deep, while others are very shallow
Breadth-first Search

- Breadth-first search treats the frontier as a **queue**
  - it always selects one of the earliest elements added to the frontier.

**Example:**
- the frontier is \([p_1, p_2, \ldots, p_r]\)
- neighbours of \(p_1\) are \(\{n_1, \ldots, n_k\}\)

**What happens?**
- \(p_1\) is selected, and tested for being a goal.
- Neighbours of \(p_1\) follow \(p_r\) at the end of the frontier.
- Thus, the frontier is now \([p_2, \ldots, p_r, n_1, \ldots, n_k]\).
- \(p_2\) is selected next.
Illustrative Graph — Breadth-first Search
Analysis of Breadth-First Search

- Is BFS complete?
  - Yes (but it wouldn’t be if the branching factor for any node was infinite)
  - In fact, BFS is guaranteed to find the path that involves the fewest arcs (why?)
- What is the time complexity, if the maximum path length is \( m \) and the maximum branching factor is \( b \)?
  - The time complexity is \( O(b^m) \): must examine every node in the tree.
  - The order in which we examine nodes (BFS or DFS) makes no difference to the worst case: search is unconstrained by the goal.
- What is the space complexity?
  - Space complexity is \( O(b^m) \): we must store the whole frontier in memory
Using Breadth-First Search

▶ When is BFS **appropriate**?
  ▶ space is not a problem
  ▶ it’s necessary to find the solution with the fewest arcs
  ▶ although all solutions may not be shallow, at least some are
  ▶ there may be infinite paths

▶ When is BFS **inappropriate**?
  ▶ space is limited
  ▶ all solutions tend to be located deep in the tree
  ▶ the branching factor is very large
Search with Costs

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

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

▶ In this setting we often don’t just want to find just any solution
  ▶ Instead, we usually want to find the solution that minimizes cost

▶ We call a search algorithm which always finds such a solution optimal
Lowest-Cost-First Search

- 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.
  - We say “a path” because there may be ties
- When all arc costs are equal, LCFS is equivalent to BFS.
- Example:
  - the frontier is $[⟨p_1, 10⟩, ⟨p_2, 5⟩, ⟨p_3, 7⟩]$ 
  - $p_2$ is the lowest-cost node in the frontier
  - neighbours of $p_2$ are $\{⟨p_9, 12⟩, ⟨p_{10}, 15⟩\}$
- What happens?
  - $p_2$ is selected, and tested for being a goal.
  - Neighbours of $p_2$ are inserted into the frontier (it doesn't matter where they go)
  - Thus, the frontier is now $[⟨p_1, 10⟩, ⟨p_9, 12⟩, ⟨p_{10}, 15⟩, ⟨p_3, 7⟩]$.
  - $p_3$ is selected next.
  - Of course, we’d really implement this as a priority queue.
Analysis of Lowest-Cost-First Search

- Is LCFS **complete**?
  - not in general: a cycle with zero or negative arc costs could be followed forever.
  - yes, as long as arc costs are strictly positive
- **What is the time complexity**, if the maximum path length is $m$ and the maximum branching factor is $b$?
  - The time complexity is $O(b^m)$: must examine every node in the tree.
  - Knowing costs doesn’t help here.
- **What is the space complexity?**
  - Space complexity is $O(b^m)$: we must store the whole frontier in memory.
- Is LCFS **optimal**?
  - Not in general. Why not?
  - Arc costs could be negative: a path that initially looks high-cost could end up getting a “refund”.
  - However, LCFS is optimal if arc costs are guaranteed to be non-negative.