CMPSCI 250: Introduction to Computation

Lecture #24: General, Breadth-First, and Depth-First Search
David Mix Barrington
26 March 2012

General, Breadth-First, and Depth-First Search

- Four Examples of Search Problems
- State Spaces, Search, and Optimization
- The Generic Search Algorithm
- When Do We Know That Generic Search Works?
- Depth-First Search
- Breadth-First Search
- Iterative Deepening Depth-First Search

Four Examples of Search Problems

- Many computational problems are **searches** over some **state space**.

- A **navigation program** is given a start location and an end location, and has a database of information about streets. It should not only find a path from start to finish, but the *best* path in terms of distance or driving time.

- A **sudoku puzzle** is a 9×9 grid where each square is to be filled with a number from 1 through 9. Some of the numbers are initially filled in, and the goal is to fill in the rest while obeying certain rules.

- The **eight queens puzzle** is to place eight chess queens on an 8×8 board so that no queen attacks another horizontally, vertically, or diagonally.

- The **Rubik's cube** can be placed in any of about 4.3×10^{19} different positions, and the goal is to return it to the start position by making legal moves.

State Spaces, Search, and Optimization

- In each of these problems there is a set of possible **states** we may be in, and a set of legal **moves** among those states. The **search problem** is to find a path from one state to another if there is one, and the **optimization problem** is to find the cheapest path according to some **cost measure**.

- At least conceptually, we can represent the state space and moves as a **directed graph**, with states as the nodes and directed edges for the moves. But it may not be possible to store the entire graph in a computer at one time. We say that the graph is **implicitly represented** if we can remember any given state and can calculate the possible moves out of it.

- How we define the state space can have an enormous impact on the difficulty of the problem. In the eight queens problem, there are over 4 billion ways to place eight queens on 64 spaces, but only 40320 that have one queen per row and one per column.

The Generic Search Algorithm

- We can define a **generic search algorithm** for any state space and set of moves. It is **underspecified** in that we won't always say what will happen, but only what might happen.

- The key data structure is the **open list**, which is a set of states that still need their neighbors to be searched. We are looking for a path from the start node s_0 to any state in a given set of **goal states**.

```
open list = {s0};
while (open list is not empty) {
   s = a state taken from the open list;
   if (s is a goal state) declare victory;
   else for (each neighbor n of s)
      add n to the open list;
   remove s from the open list;}
declare defeat;
```

When Do We Know That Generic Search Works?

- We would like our search to declare victory whenever a path exists from s_0 to any goal state, and to declare defeat whenever no such path exists. When can we count on this? Here are four lemmas, proved carefully in the text.

- If the search declares victory, we can prove by induction that a path exists.

- If the search declares defeat, we can prove that there is no such path. (We use the contrapositive method -- if a path exists we won't declare defeat before we find it.)

- If a path exists, *and* every state added to the open list is eventually removed from it, the search will eventually terminate and declare victory.

- If no path exists, *and* there are only finitely many states in the search space, *and* each state enters the open list only finitely many times, then the search will eventually terminate and declare defeat.

Polynomial Versus Exponential Search

- A search algorithm that will *eventually* find a path to its goal is not much use if it takes too long to do so. We'd like to be able to estimate the number of steps we will need. But we may not even know the size of the state space if it is implicitly represented. (Sometimes we just have an **upper bound** on it.)

- Mathematical analysis of running times is usually for **parametrized** problems, where there is some size factor n, like the size of the space or the maximum length of paths that interest us.

- A key distinction is between time functions that are **polynomial** in n, such as n^2 or n^{10}, and functions that are **exponential** in n such as 2^n. The latter are much worse and usually become prohibitive for even very small n.

- Exhaustive search of all paths is usually exponential -- if each state has d neighbors there are about d^n paths of length n.

Depth-First Search

- Our generic algorithm didn't specify *which* state we take off the open list when we need a new one. We could always take off the one that was *most recently* put on, making the open list a **Last-In-First-Out** structure or a **stack**.

- Another issue is whether we can recognize states that we have already explored when we see them again. If we can store the whole graph we can just mark these nodes, and if not we could possibly keep a **closed list**. But in general space is more expensive than time when we search huge spaces.

- Depth-first search is **greedy** in that it explores all the consequences of its first choice before considering alternatives to it. If our search is totally blind, we could even get stuck in an infinite cycle and never complete the search.

- In a **directed acyclic graph** we are at least guaranteed to finish the search.

A Depth-First Search Example

- Consider a Manhattan grid where we start at the southwest corner and edges are directed north and east. Let's look at what happens if our state space is the points whose "Manhattan distance" from the start is at most 4, and there are no goal nodes. (This is the worst case for the time of a search.)

- We begin by putting (1, 0), (2, 0), (3, 0), and (4, 0) on the stack. We pop (4, 0) off as it has no neighbors, and return to (3, 0) to check (3, 1). When that fails we return to (2, 0) to check (2, 1), which runs searches of (3, 1) and (2, 2) -- it doesn't know that it has already checked (3, 1). We then return to (1, 0) and search (1, 1) and all its descendants, then return to (0, 0) and search (0, 1) and all of *its* descendants.

- All in all we searched each of the $2^4 = 16$ paths even though there were only five nodes with no descendants. With no recognition of previously seen nodes, we will search 2^n paths if we search the grid up to distance n.

Breadth-First Search

- The other natural way to manage the open list is with a **First-In-First-Out** structure, or a **queue**. This has a number of advantages.

- We will find a path if one exists, as long as each node has only finitely many neighbors. This is because we put all nodes at distance 1 on the queue, then distance 2, then distance 3, and so on. Once we reach the distance of the nearest goal node, we will look at *all* nodes at that distance and thus find that goal node. Thus we find the *shortest* path, in terms of number of edges. (If different edges have different costs, this may not be the real cheapest path.)

- Depth-first search might be much faster if its greedy search succeeds immediately -- breadth-first search *must* check all paths shorter than the right one. BFS also uses much more memory in general, as all the nodes at a given distance are stored on the queue at once.

- Without recognizing seen nodes, BFS and DFS take about the same time on our example as they put a node on the open list once for each path to it.

Iterative Deepening Depth-First Search

- When we can't recognize seen nodes, a hybrid approach between DFS and BFS, called **iterative deepening DFS**, can combine the advantages of both.

- The idea is to carry out a DFS but **truncate** it at distance 1. If that fails, DFS again truncating to distance 2, then distance 3, and so on. Like BFS, this is guaranteed to find a shortest path in terms of number of edges.

- We only need to keep a stack rather than a queue. If the graph has degree d, the stack for the distance-k DFS will have at most k nodes on it, while the queue for the corresponding BFS might have as many as d^n nodes on it.

- We appear to be wasting time by doing all the shorter searches before we discover the right distance. But since these searches get exponentially longer with k, the distance-k one takes more time than all the others put together. So we waste only a small fraction of the time for the right search.