CS410 Lecture Notes

September 2024

1 Blind Search (BFS + DFS)

1.1 Overview

Introduction to problem formulation (how do we formally define problems in terms of states and transitions?) and blind search algorithms (DFS + BFS). You actually solve these kinds of problems everyday! For example, you took a specific route to get from your dorm to List 120 for class - how did you pick which turns to make, or which roads to go down? There are multiple ways to get to your destination, but you picked one - this is a search problem!

1.2 Definition of a Search Problem

A search problem contains 4 major parts, {X, S, G, T}, where:

$$\begin{aligned} X &: \text{set of states} \\ S &: \text{set of start states} \\ G &: \text{set of goal states} \\ T &: \text{transition model} \end{aligned}$$

X represents the set of all possible states we can be in. It follows that S and G will be subsets of X - this makes sense since the states at which we can start or end at will be part of the set of all possible states. The transition model maps a single state to a set of possible successor states - or in other words, defines what are the possible "next moves" we can make from the state we are currently in.

Note: it is possible to have multiple start states and multiple goal states, but for the problem of path finding, a single start makes more sense (since we want to get from place A to place B). If we want to start from a different place - say, the Rock instead of your dorm - then we can just create another search problem with a different S! We will go over different scenarios where you might have more than one start state in future lectures.

Let's define a search problem in the context of **Google Maps** - they are essentially solving the problem of path planning. We can define the problem as follows:

$$\begin{aligned} X &: \text{all intersections between roads} \\ S &: \text{where I am right now (ex. List 120)} \\ G &: \text{any set of restaurants (ex. Chinatown and DenDen)} \\ T &: \text{1 block in either of the 4 directions (North, East, South, West)} \end{aligned}$$

In this case, I am stating that I would like to find a path from where I am now (List 120) to any of 2 restaurants: DenDen or Chinatown. If I reach either of those 2 restaurants, then the goal is satisfied and my problem has been solved. I can break down my path into blocks traveled in each of the 4 directions, which is modeled by my transition function (T). This makes sense since to get from one intersection (or state) to the next, we have to travel 1 block.

Let's do another, more interesting example: Donald Knuth's problem of turning 4 into any number with any combination of these operations: $\sqrt{x}$, $x!$, or $\lfloor x \rfloor$. We can then define the parameters of our search problem as follows:

$$X : \mathbb{R}^{+}\text{(i.e. the set of all positive, real numbers)}$$
$$S : 4$$
$$G : 5 \text{ (as an example)}$$
$$T : T \rightarrow x : \{\sqrt{x}, x!, \lfloor x \rfloor\}$$

1.3 Search Algorithms

Now that we know how to define a search problem, we need to learn how to solve them. This can be done using popular search algorithms: BFS and DFS.

Prior to defining these algorithms, we need define a **frontier**: the set of nodes at the edge of a search (or the nodes that we will search next). These can also be called "open" nodes (or "unvisited" nodes), while nodes that we have seen before are called "closed" or "visited" nodes.

The general outline of a search problem is as follows:

Algorithm 1 General Search Problem

```
1: Procedure SEARCH($X$, $S$, $G$, $T$):
2: Define $O$: a set of "open" nodes (a.k.a the frontier)
3: Add $S$ (start state) to $O$
4: while $O$ not empty do
5:    remove $n$ from $O$ ($n$ is an open node)
6:    if $n \in G$ then
7:       return $n$
8:    end if
9:    append successors of $n$, $T(n)$, to $O$
10: end while
11: End Procedure
```

1.3.1 Breadth First Search (BFS)

In BFS, the above psuedocode applies, but we take O as a queue. A queue follows First-In First-Out (FIFO). You can think of this as a "first-come first-serve" data structure - this means that we remove/process successor states from the beginning of O and add successor states to the end of O. As a reminder, the algorithm works on a tree of nodes and their successors as follows:

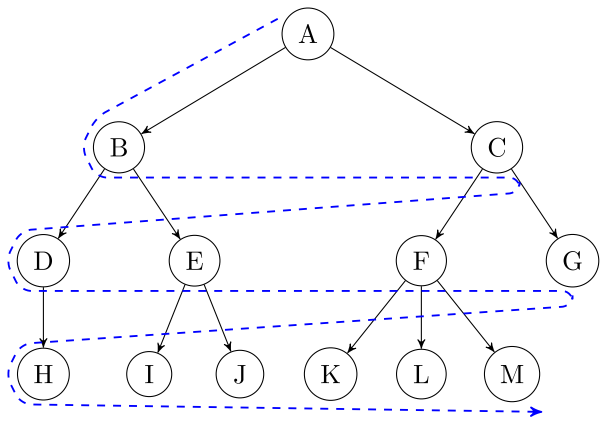

Figure 1: BFS traversal

1.4 Performance measures for search algorithms

We care about measuring performance in four ways:

1. Runtime: How long it will take to find a goal (number of states expanded)
2. Space Complexity: Length of open set (number of states in frontier)
3. Completeness: If a solution exists, does the algorithm find it?
4. Optimality: Does the algorithm return an optimal solution (shortest path, in most cases)

For these analyses, we will consider search problems on a tree with maximum depth d and branching factor b. BFS' performance is:

1. Runtime: $O(b^d)$, at worst case BFS has to expand every node in the tree.
2. Space Complexity: $O(b^d)$, at worst case BFS stores the entire final depth of the tree in the open set, which is $O(b^d)$
3. Completeness: BFS *is* complete. If a solution exists, BFS will eventually explore that depth and find it.
4. Optimality: BFS does return shortest paths since it finds the goal at the shortest depth.

2 Lecture 2