

Basic Search

CS 221

Section 1

September 25, 2009

1 Introduction

Today we will discuss some basic blind search algorithms and work some example problems involving them. These algorithms should be a review for most of you, as you have probably seen them in previous classes.

2 The problem

Today we will focus on basic search on a directed graph.

A directed graph G consists of:

- N , the set of nodes (vertices) in the graph
- $E \subseteq N \times N$, the set of edges in the graph

For a search problem on a graph we are trying to find a path in this graph from some start node s to some goal node g . Altogether, then our search problem consists of:

- A graph $G = (N, E)$
- A start node $s \in N$
- A goal node $g \in N$

A solution to a search problem is a path from s to g . This can be represented as an ordered list of nodes (n_1, n_2, \dots, n_k) , where $n_1 = s$, $n_k = g$, and each pair $(n_i, n_{i+1}) \in E$.

2.1 Terminology

- **To expand**, or **investigate**, a node n is to generate the set of all nodes which are connected to n by a directed edge. (i.e. the set of all nodes k such that $(n, k) \in E$). The **successor function** is a function $N \mapsto 2^N$ which generates this set for each node.
- A node k is **generated**, or **discovered**, when a node n to which it is connected (i.e. (n, k) is in E) is expanded.
- **The fringe** is the data structure we use to store all of the nodes that have been generated but not yet expanded.

2.2 Basic search idea

We begin with only the start node s in the fringe. We then expand s , adding each of the successors of s to the fringe. A node in the fringe is selected, removed and expanded. When a node is removed from the fringe it is checked to see if it is the goal g . If it is we stop our search.

2.3 Search strategies

A search strategy simply determines which node in the fringe to expand next. Today we will go over some of the more basic ones.

2.4 Measuring performance

First we digress slightly to discuss the things we desire from a search algorithm. What are the ways we measure the performance of a search strategy or algorithm?

- **Completeness**: will the algorithm find a solution if one exists?
- **Optimality**: will the algorithm find the optimal solution? (lowest path cost among all solutions)
- **Time complexity**: how long does it take to find a solution?
- **Space complexity**: how much memory is needed to perform the search?

Some properties of the search space we will use to answer these questions are as follows:

- **branching factor**: b maximum number of successors of any node
- **depth of solution**: d is the minimum number of steps to get to the goal
- **maximal path length in the space**: what is the longest path we could follow?

3 Blind search

We call the following search algorithms **blind** or **uniformed** because the algorithm has no extra information about the nodes. It can only tell goal nodes from non-goal nodes

3.1 Breadth-first search

Breadth-first search first selects nodes in the fringe that are the fewest steps away from the start node. All nodes that are m steps away from s are expanded before those nodes that are $m + 1$ steps away. One way to accomplish this is to implement the fringe with a FIFO queue. We extract from the fringe the node that has been in it the longest.

3.1.1 Evaluation

- **Complete?**
Yes, provided b is finite. We examine all nodes up to and including depth d .
- **Optimal?**
Yes, if all edges have the same cost.
- **Time complexity:**
Let's count all of the nodes that are generated for a goal at level d .

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

- **Space complexity:**
Same as time complexity. Every node generated must stay in memory because it is either in the fringe, or is the ancestor of a fringe node.

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabyte
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

BFS search example

3.2 Uniform cost search

Breadth-first is optimal when path costs are equal, since it finds the goal node which is the fewest steps from the start node. **Uniform-cost search** uses the same idea in situations where path costs may differ (we assume that all costs are non-negative). Where BFS expands the node with the lowest depth, UCS expands the node with the lowest cumulative path cost. This will guarantee that if it expands the goal node then it has found the optimal solution. One way to implement UCS is with a priority-queue, where nodes are ordered based on their g -value, or the cost of the path to them so far. When we extract a node from the fringe, we extract the node with the minimum g -value.

3.2.1 Evaluation

- **Complete?**
Yes, if all path costs are greater than or equal to some positive ϵ and b is finite. If we have an infinite path with edge costs of 0, then we would go down it forever and never explore other options. Having a minimum path cost of ϵ avoids this.
- **Optimal?**
Yes. When we expand the goal node we know that all other possible paths have longer length, so the path found must be optimal.
- **Time and space complexity:**
 $O(b^{1+\lceil C^*/\epsilon \rceil})$, where C^* is cost of optimal solution, and ϵ is the smallest action cost. This can be more than breadth-first, but if all actions have the same cost, it reduces to the same complexity as BFS.

3.3 Depth-first search

Depth first search always expands the *deepest* node in the fringe. This can be implemented using a LIFO queue (a stack) for the fringe. In this manner we will explore all of a node's descendants before its siblings. Because of this, once we move on to a node's siblings, we can discard this node, since we don't have any of its descendants in the fringe and so it cannot lie on the solution path.

3.3.1 Evaluation

- **Complete?**
No. Can go down infinite paths
- **Optimal?**
No. Might be shorter path to goal.
- **Time complexity:**
 $O(b^m)$ in the worst case

- **Space complexity:**

$O(bm)$ (backtracking search can get this down to $O(m)$).

To compare with the BFS example shown earlier, DFS at depth 12 requires 118 kilobytes of memory, while BFS required 10 petabytes, a factor of 10 billion times more space.

3.4 Depth-limited search

So far, it seems there is hardly anything good about DFS, except for its good space complexity. Can we address any of these shortcomings? What if we knew d ? Then we could only search to that depth. In general, we can determine a depth limit l and never generate any nodes beyond that depth. We must be careful setting l , since if $l < d$ we won't find a solution. This changes our time complexity to $O(b^l)$, and our space complexity to $O(bl)$

But how do we know what limit to place on the depth?

3.5 Iterative-deepening search

What if we gradually increase the depth-limit, running a depth-limited search at each depth? At first, this is a terrifying idea! It seems so wasteful, since we are throwing away entire searches. This works, though, since most nodes in a tree with near-constant branching factor are in the bottom level. Iterative deepening is actually faster than BFS, since it doesn't generate the nodes at level $d + 1$.

It is complete \Rightarrow time = $O(b^d)$, space = $O(bd)$, and it is also optimal if steps costs are all identical.

For example, comparing the number of nodes generated by IDS and BFS on a graph search problem with $b = 10$ and $d = 5$, we have:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

In general, IDS is the preferred uniformed search method for a large search space with unknown solution depth.

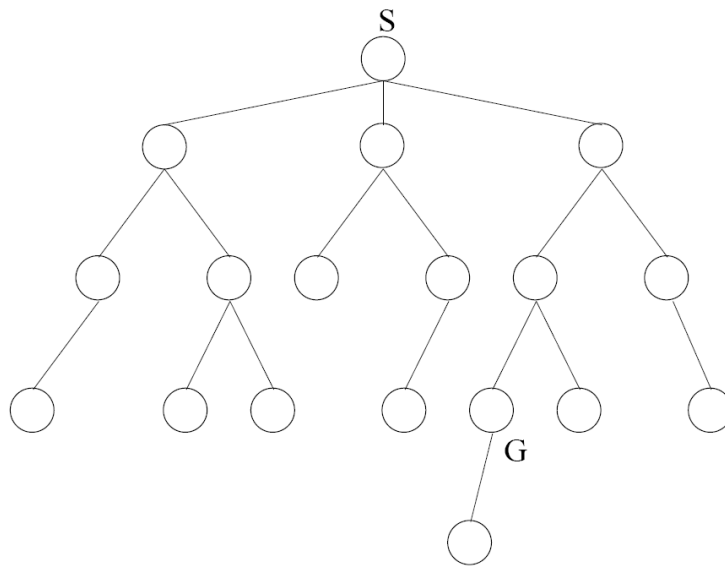
4 Example questions

1. In this problem you will implement different search strategies on a given search tree. The start state is denoted by S and the goal state by G . Number the nodes in the tree according to the order in which they will be expanded. (Recall that a node is expanded when it is removed from the list of nodes, checked for goalness, and its children are inserted into the list.) Do not number a node if it is not expanded in the search.

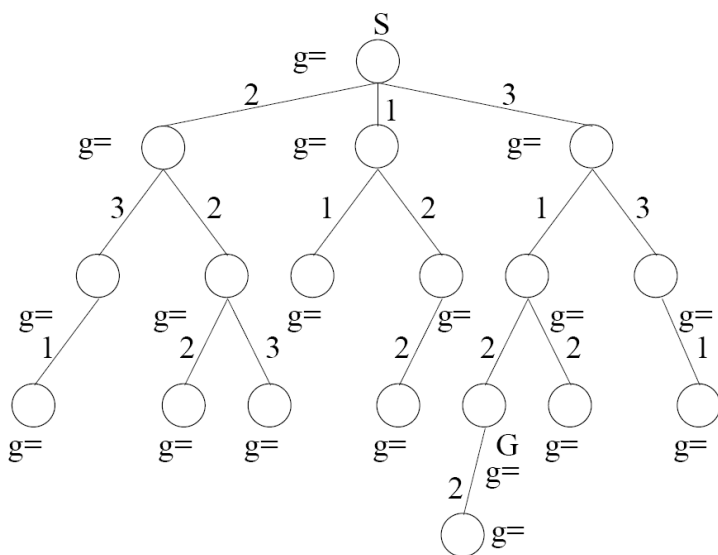
Assume that the children of a node are inserted into the list in left to right order, and that nodes of equal priority are extracted from the list in FIFO order.

Repeat this for each of the following search strategies:

- (a) Breadth First Search
- (b) Depth First Search
- (c) Iterative Deepening Search (Hint: here, a node may have multiple labels.)



- (d) Uniform Cost Search, where the costs of the edges are as specified in the following tree. Here, also write down the g -value of the different nodes in the tree. You only need to write down g -values for nodes that are inserted into the list.

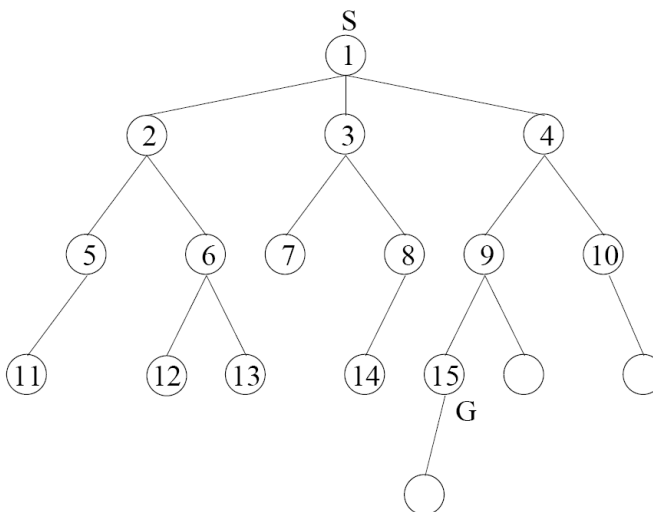


2. Describe a search space in which iterative deepening search performs much worse than depth-first search.

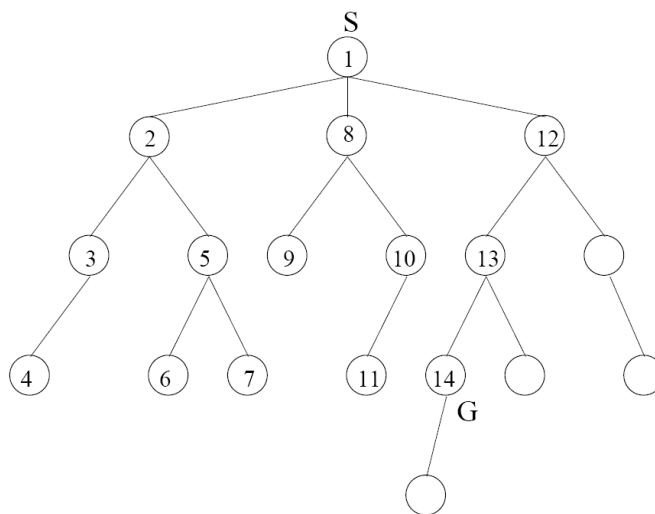
5 Example answers

1. Search strategies:

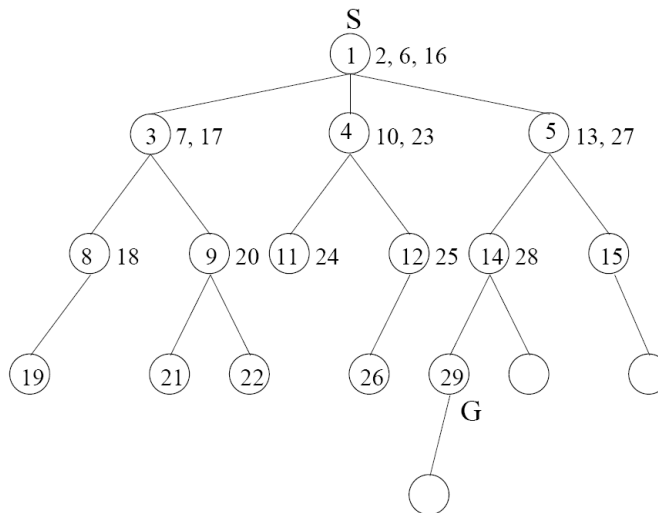
(a) Breadth First Search.



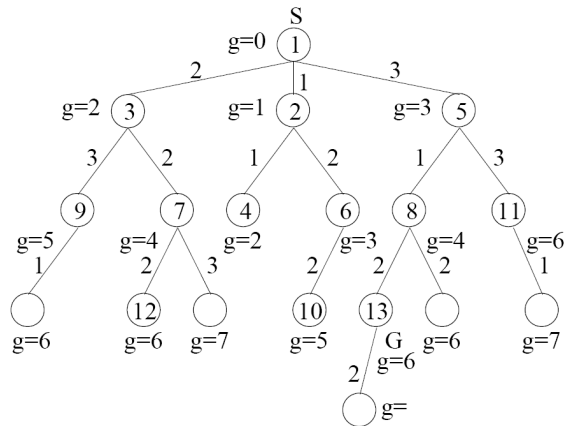
(b) Depth First Search.



(c) Iterative Deepening. (Hint: Here, a node may have multiple labels.)



- (d) Uniform Cost Search, where the costs of the edges are as specified on the tree. Here, also write down the g -value of the different nodes in the tree. You only need to write down g -values for nodes that are inserted into the list.



2. Describe a search space in which iterative deepening search performs much worse than depth-first search.

Answer: Consider a space with a branching factor of 1 at every move, and a solution at depth n . A depth-first search will take exactly n expansions to find a solution, while an iterative deepening search will take $O(n^2)$.