

Blind Search

CS 121

Lecture 2B

Chris Archibald

July 29, 2009

1 Recap: Search Problem

A **search problem** has:

- States
- Initial state
- Successor function
- Goal test
- Path cost

With a common formulation for search problems, now we need to **solve** them. Chapter 3 talks about search techniques that use an explicit **search tree**.

First let's see an example

2 Example: Search in Romania

For this example

- States: The cities in Romania
- Initial state: *Arad*
- Successor function: The set of cities that we can drive to from this city
- Goal test: *Bucharest*
- Path cost: ignore for now

First test to see if Arad is the goal. It isn't, so we move on.

- Arad \Rightarrow {Sibiu, Timisoara, Zerind }
- Sibiu \Rightarrow {Arad, Fagaras, Oradea, Rimnicu Vilcea }
- Timisoara \Rightarrow {Arad, Lugoj}
- Zerind \Rightarrow {Arad, Oradea}

2.1 Terminology

To **expand a node** is to apply the successor function to a state and **generate** a new set of states **A node is generated** when it is the successor to an expanded node

2.2 Distinction between Nodes and States

A **node** is a data structure which contains

- State
- Parent-node
- action
- path-cost
- depth

A **state** corresponds to a configuration of the real world. Two nodes can contain the same world state, if that state is generated via two different search paths.

2.3 The fringe

This stores all of the nodes that have been generated, but not expanded. An element of the fringe is a leaf node.

Q: Is the fringe equal to the set of all leaf nodes?

2.4 Search

A search strategy simply determines which node in the fringe to expand next. We can assume that the fringe is a queue.

2.5 Measuring performance

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?

Factors =

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

3 Blind search

Blind or uniformed, because the algorithm has no more information than just the state description. Can only tell goal states from non-goal states.

3.1 Breadth-first search

The fringe is a FIFO queue. Insert generated nodes at the end of the queue, they are extracted from the front of the queue

Run on example tree

3.1.1 Evaluation

- **Completeness:** will the algorithm find a solution if one exists?
Yes, provided b is finite. We examine all nodes up to and including depth d .
- **Optimality:** will the algorithm find the optimal solution? (lowest path cost among all solutions)
Not necessarily, but yes if actions all have same cost. It does find the shallowest solution (fewest actions).
- **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:** how much memory is needed to perform the search? 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

Assumptions: $b = 10$, 10,000 nodes per second, 1000 bytes per node

3.1.2 Lesson: Time

Exponential-complexity search problems cannot be solved by uniformed search methods for any but the smallest instances

3.1.3 Uniform cost search

Breadth-first is optimal when path costs are equal, it finds the shallowest goal node. **uniform-cost search** expands the node with the lowest path cost. This will guarantee that if it finds a goal it is the optimal solution.

Q: Is it complete? Why? Is it optimal?

If all path costs are greater than ϵ then it is complete and optimal.

Complexity is $O(b^{1+\lceil C^*/\epsilon \rceil})$, where C^* is cost of optimal solution, and ϵ is the smallest action cost. Can be more than breadth-first, or the same, if actions have the same cost (obviously).

3.2 Depth-first search

Depth first search uses LIFO queue, searches the deepest node in the fringe.

Run on example tree

Notice how we can discard nodes, because we don't have any of their descendants in the fringe, they cannot lie on the solution path.

3.2.1 Evaluation

- **Completeness:** will the algorithm find a solution if one exists?
No. Can go down infinite paths
- **Optimality:** will the algorithm find the optimal solution? (lowest path cost among all solutions)
No. Might be shallower goal down other path.

- **Time complexity:**
 $O(b^m)$ in the worst case
- **Space complexity:** how much memory is needed to perform the search?
 $O(bm)$ (backtracking search can get this down to $O(m)$).

DFS at depth 12 requires 118 kilobytes, while BFS required 10 petabytes, a factor of 10 billion times less space.

3.3 Depth-limited search

Q: How can we fix some of depth-first searches problems?

Determine a depth limit l . Be careful setting l . Time = $O(b^l)$, space = $O(bl)$

Maximal depth of solution gives us **diameter** of the state space. In general hard to come up with depth limit.

Not complete (what if $l < d$?) Not optimal, if $l > d$. Can view depth first search as depth-limited with $l = \infty$

3.4 Iterative-deepening search

What if we gradually increase the depth-limit?

At first, terrifying idea! It seems so wasteful. But, 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 - time = $O(b^d)$, space = $O(bd)$, also optimal if steps costs are all identical.

For example, compare to BFS 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.

3.5 Bidirectional search

Run two searches, one from start, one from goal, check for meeting. Motivated by $2b^{d/2}$ being much smaller than b^d . Check each node when expanded to see if it is in the fringe of the other search.

Complication is the required **Predecessors** function. Some are hard to reverse, or hard to compute.

Also, when there are multiple goals (8-queens) where do we start?

Or what if the goal is just implicit (like checkmate in chess)?

Q: Would IDS + bidirectional search be a good combination?

4 Repeated states

This can be a big problem, and render some problems not solvable when they should be solvable

Consider a grid, where each state has 4 successors, the search tree has 4^d leaves, but there are only $2d^2$ distinct states within d steps of any state. For $d = 20$ this means the tree has a trillion nodes, but there are only about 800 distinct states.

Depth-first only can compare with other nodes on that path, but can still have problems (see figure 3.18)

One basic idea is to keep a list of already visited states, check each state you visit to ensure that you haven't already seen it. This really impacts the memory requirements for the DFS style algorithms

Have to be careful about which path we keep

5 Conclusion

Today: We learned how to formulate a problem as a search problem, and then some algorithms for solving these problems, given no additional information about the states/space.

Should be familiar with the following and their properties:

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

Next time we will be discussing heuristic search, where the algorithm can focus its search on more promising states and directions. See you all then!

6 Implementation

Basic algorithm

1. Insert **START** into **FRINGE**
2. while **FRINGE** is not empty
 - (a) get *node* from **FRINGE** according to strategy
 - (b) (2) Is **GOAL-TEST**(*node.state*) true? Yes - then done
 - (c) for each $s \in \text{SUCC}(\textit{node})$
 - i. Create a new node n' as a child of *node*, with state s
 - ii. (1) Is **GOAL-TEST**(s) true? Yes - then done
 - iii. Insert n' into **FRINGE**