# Lecture 4: Uninformed Search

*9/8/2016*

## Summary

In this lecture, we compared some commonly used *uninformed search strategies* (i.e., they don't make use of any special knowledge or information about the problem).

## Recap from last time

We presented the master recipe for implementing search algorithms. We showed both the tree-search variant (no duplicate state check) and the graph-search variant (extra duplicate state checking, shown in green in the following pseudocode).

**Function** graph-search (problem): returns a *solution*, or *fail*

Init frontier list with root node (has init state of problem) Init explored list to empty.

**loop do**:

**1. If** frontier is empty : **then return** *fail*

2. Choose & remove a node from frontier

**3. if** the node contains goal state :

**then return** *solution* (*i.e., path from current node back to root*)

4. Add the state in that node to explored

5. Expand that node, add those children nodes points to successor states not already in explored (or already pointed to by another node in frontier) to frontier

Note: frontier and explored are different types (one store search nodes, the other store states)

Question: what data structure should we use for implementing the explored list?

## Search Strategies

We will see quite a few strategies. We will compare them by considering their properties in addressing these four concerns:

- Completeness
- Optimality
- Time complexity
- Space complexity

## Uninformed Search Strategies

We will start with uninformed search strategies -- that is, they don't take into account any additional information about the problem. They basically work the same way whether we are doing the jug problem or the 8-puzzle problem.

- Breadth First Search (BFS)
  - Frontier list is a FIFO queue
  - It's both optimal and complete (why?)
  - But it's exponential in both time and space complexity.
    - Question: Which is worse? (Answer: space)
  - Digression on geometric series (this came up in class and I didn't remember the formula):
    - $1+b+b^2+b^3+...+b^d = (1 - b^{d+1})/ (1 - b)$, which we rearrange to: $(b^{d+1}-1)/(b-1)$ to show that the sum is approximately $b^d$, especially when the branching factor is big.

    The number of nodes at the fringe dominates the total sum of internal nodes. (This is important for our argument for search strategies that rely on iterative deepening, as we'll see in a bit)
- Depth First Search (DFS)
  - Frontier list is a LIFO stack
  - Neither complete nor optimal in general. Why?
  - Efficient in space. Why?
  - Question: Does it make sense to use graph-search (i.e., have an explored list) for the DFS family?
- Depth-limited DFS
  - Set a maximum cut-off so search won't go down one branch infinitely.
  - Like DFS, it's neither complete nor optimal. So what's good about it?
    - For what types of problems would we use Depth-Limited DFS instead of regular DFS?
- Iterative deepening DFS
  - iteratively running depth-limited DFS (i.e., first, set cut-off = 1 and run depth-limited DFS. If we don't find a solution, set cut-off = 2 and run depth-limited DFS. If we don't find a solution, set cut-off = 3 etc.
  - Why is it complete?
  - Why is it optimal?
  - Surprisingly, its time complexity is arguably no worse than BFS
    - the frontier of the tree is so much bigger than all the rest of the tree put together:
      - First time: 1
      - Second time: $1 + b$
      - …
      - Penultimate time (cut-off at d-1): $1 + b + … + b^{(d-1)}$
      - Final time (cut-off at d): $1 + b + … + b^{(d-1)} + b^d$
      - Adding all times together: $d + (d-1)b + … + 2b^{(d-1)} + b^d$
        The final $b^d$ term still dominates so it's still $O(b^d)$.

        - In practice, it might take longer than BFS because it is doing more repeated work; on the other hand, because its memory footprint is small, you don't lose time doing memory allocation stuff (which also could slow the machine down).

- Bidirectional
    - Run 2 searches: one from the start state forward, and one from the goal state backward, and when they meet up, you have a solution.
    - Complete if at least one of the searches is complete. Why?
    - Optimal for reasonable search choices (what are bad choices?)
    - Not always applicable:
        - Might be hard to figure out the previous state from the current one.
        - Multiple goal states
        - Implicit goal states

Summary Table (from AIMA)

| Criterion | BFS | DFS | Depth-Limited | Iterative Deepening DFS | Bidirectional (with BFS) |
|---|---|---|---|---|---|
| Complete? | Yes | No | No | Yes | Yes |
| Time | $O(b^d)$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^d)$ | O(bm) | O(bl) | O(bd) | $O(b^{d/2})$ |
| Optimal? | Yes | No | No | Yes | Yes |

b: branching factor
d: depth where optimal solution is found
m: maximum depth of tree (potentially infinite)
l: cut-off limit for depth-limited search

## Corresponding Readings

- Chapter 3.4 of Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach. 3rd ed. Prentice Hall.