Blind Search Methods

Kris Beevers
Intro to AI 9/11/03
Ch. 3.1-3.5

Overview

- "Uninformed" algorithms: given no information about problem other than its definition
- Later will deal with "informed" algorithms (Ch. 4 Heuristic Search)
- *Solving a problem*
  1. Formulate **goal** (decide what the *goal state* is)
  2. Formulate **problem** (decide what *actions* and *states* to consider, given a goal)
  3. **Search** for the goal

Problem Formulation

- **initial state** (e.g. $In(Troy)$)
- **actions** (generally transform the agent from one state to another); commonly, a *successor function* maps the set of states onto a set of (action, successor-state) pairs
- **cost**: actions generally have some positive (or at least non-negative) cost
- **state space**: set of all states reachable from the initial state; forms a graph (nodes are states, edges are actions); *this is not the same as a search tree*
- **path** through state space: sequence of actions
- **objective**: minimum cost path from start state to goal state (optimal solution)
- state space not the same as **search tree**; search tree generated by state space/successor function during search

- states not same as **nodes**: nodes contain a state plus other information (like pointers to children or parents)

Example Problems

- 8-puzzle/15-puzzle (sliding block puzzles)
- 8-queens problem (place 8 queens on board such that no queen can attack another)
- route-finding problems (can generalize to robot motion planning)
- touring problems (traveling salesman)
- VLSI layout (placing components on a chip to minimize area, circuit delays, etc.)
- Assembly sequencing
- Protein folding

Properties of Searches

- **optimality**: will an algorithm find the *lowest cost* path to the goal?
- **completeness**: will an algorithm find a path to the goal if one exists?
  - used to define as "return failure otherwise"
  - if there is no solution, and the algorithm cannot detect this (detect repeated states), then it creates an infinite search tree and *no result will ever be returned*
- **time complexity**: run time of search (e.g. number of nodes generated during search)
- **space complexity**: memory requirements

Blind Search Algorithms

We discuss six algorithms for blind search. Generally, we use a queue formulation of the algorithms. First a few definitions:

- b: branching factor (or average branching factor); number of successors to any node
- d: minimum solution depth in search tree
- m: maximum depth of search tree

Depth-first Search (DFS)

- Use slide (cover up the bottom and discuss it first!)
- Draw a figure
- Possible problem: infinite search (infinite depth tree)

Optimal?	No
Complete?	No
Time complexity	$O(b^m)$
Space complexity	$O(mb)$

Breadth-first Search (BFS)

- Use slide (cover up the bottom and discuss it first!)
- Draw a figure

Optimal?	Yes
Complete?	Yes
Time complexity	$O(b^d)$
Space complexity	$O(b^d)$

Uniform Cost Search (UC)

- Instead of expanding shallowest node (BFS) or deepest node (DFS), expand node with *lowest path cost*
- Draw a figure
- Problem: gets stuck in an infinite loop if it ever expands a node with a zero-cost action leading back to the same state (i.e. a $NoOp$ action)
- Solution: require the cost of every step to be greater than or equal to some small positive constant ϵ
- Given this condition, UC is optimal and complete
- Let C^* be cost of optimal solution; then, worst-case time/space complexity is [see table]; this can be much greater than b^d (UC often explores large trees of small steps before exploring paths with large, perhaps more useful steps)

Optimal?	Yes
Complete?	Yes
Time complexity	$O(b^{\lceil C^*, \epsilon \rceil})$
Space complexity	$O(b^{\lceil C^*, \epsilon \rceil})$

Depth-limited Search (DL)

- DFS with a predetermined depth limit l
- Treat nodes at depth l as if they have no successors
- Solves infinite path problem of DFS
- But what if the shallowest goal is at depth d such that $l < d$?
- Also non-optimal if $l > d$
- Sometimes, can base depth limits on knowledge of the problem (e.g. if we know there are only 10 states, $l = 9$ makes sense)
- But usually, don't know a good l until we've solved the problem

Optimal?	No
Complete?	No
Time complexity	$O(b^{(l+1)})$
Space complexity	$O(bl)$

Iterative Deepening Search (ID)

- Combine depth-limiting and DFS to iteratively find the best depth limit
- Gradually increase depth limit (0, 1, 2, etc.) until goal is found (occurs when depth limit reaches d)
- Figure on p. 79
- Analysis:

  Not very costly to generate nodes more than once, assuming same (or nearly the same) branching factor at every level. This is because most of the nodes are in the bottom level.

  Nodes on bottom level (depth d) generated once, depth $d-1$ generated twice, etc. Children of root generated d times. So, total number of nodes generated is

  $$N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d$$

  so time complexity is $O(b^d)$

  Compare this to BFS: BFS generates some nodes at depth $d+1$ (children of nodes being checked), IDS doesn't.

  $$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

  So, if $b = 10$ and $d = 5$, $N(IDS) = 123,450$ and $N(BFS) = 1,111,100$.

- Prefer ID when search space is large and depth of solution is unknown.

Optimal?	Yes
Complete?	Yes
Time complexity	$O(b^d)$
Space complexity	$O(bd)$

Bidirectional Search (BD)

- Two simultaneous (usually BFS) searches (one from start state, other from goal)
- Stop when the two searches meet in the middle
- Check each state when it is expanded to see if it is on the "fringe" of the other search tree
- Assume this membership checking can be done in $O(1)$ with a hash table
- Must keep at least one search tree in memory for the checking
- Requires that we can search backward from goal! Simplest when actions are reversible (e.g. $Pred(n) = Succ(n)$), but not always the case!

Optimal?	Yes
Complete?	Yes
Time complexity	$O(b^{d/2})$
Space complexity	$O(b^{d/2})$

Summary

	Optimal	**Complete**	**Time**	**Space**
BFS	Yes	Yes	$O(b^{(d+1)})$	$O(b^{(d+1)})$
UC	Yes	Yes	$O(b^{\lceil C^*,\epsilon \rceil})$	$O(b^{\lceil C^*,\epsilon \rceil})$
DFS	No	No	$O(b^m)$	$O(bm)$
DL	No	No	$O(b^{(l+1)})$	$O(bl)$
ID	Yes	Yes	$O(b^d)$	$O(bd)$
BD	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

Side Notes

- BFS used only on graphs where action cost is always the same, otherwise use UC search
- DFS not complete only because it can become lost in an infinite depth search tree

- DL search not complete only because in general a solution is not guaranteed to exist at $d \leq l$
- BD time/space complexity assumes both directions use BFS

Avoiding Repeated States

Three approaches, in order of increasing computational requirements and effectiveness:

1. Don't go back to parent (avoids cycles of size 2)
2. Don't go back to any ancestor (avoids cycles)
3. Don't revisit a state (i.e. don't explore the children of a state twice). *You must remember all states you have visited.* Can implement this in an *open/closed list* formulation: if current node is in closed list, discard it instead of expanding it. What are the implications of this on time and space requirements? (Proportional to size of the state space.)