50.021 -AI

Alex

Week 08: Search

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Search

Goal:

- we are in a start state
- we want to reach a goal state

Examples: in the city Arad, goal: to be in Bukarest

We want an algorithm that takes in the start state, and outputs us: a set of actions that lead from the start state to the goal.

What do we need for a search algorithm?

- a start state
- a test to know when we are in a goal state
- a model of what states can be, and what actions can be - how to model those?
- a successor function: given a state, and an action, how will the state change under the action? what will be the new state? ...

  Example: move left will move the bot in to the left cell relative to the current position, go from Arad to Zerind will change state from Arad to Zerind.
- possibly a cost for an action applied to a state

Tree Search

We will wrap states into nodes of a tree or graph. A node contains information about the state, but also other properties like: the set of actions chosen so far, or the current cost of the total path that leads from the start state to the state encoded in the node.

Often it is sufficient to encode a pointer to the parent and the last action taken that leads to a path. Why? Once we reached a goal state, we can just follow up all the pointers backwards to the start state. This gives us the sequence of paths.

Nodes are wrappers for the states of the world with additional information useful for the search.

- maintain a set called fringe/frontier/agenda
- initialize the fringe to contain only a node encoding the start state
- do an endless while(1==1) loop:
  - if the fringe is empty, return failure of the algorithm. No goal was found, and no state to expand anymore. can't do nothing!
  - otherwise: use a search strategy to select a node from the fringe
  - the selected node gets removed from the fringe
  - if the state of the selected node is a goal state, return success, and the path of actions
  - otherwise: *expand the state into a set of new states*: this means: loop over all actions that can be applied to the state, for every action
    - use the sucessor function to determine what will be the new state given the action and the old state
    - create a node from this new state
    - add the node to the frontier

At every step of the algorithm, when expanding a node is finished, then the fringe corresponds to the leaves in the search tree.

Uninformed Strategies

they are called uninformed because they use only

- BFS
- UCS
- DFS
- depth-limited search
- iterative deepening

some assumptions:

- b is the branching factor - every node has at most b successor nodes
- m is the maximal depth of the tree (can be infinite $m = \infty$)
- d is the depth of the optimal solution

BFS

Idea: expand the shallowest node first, that is the node with the smallest depth in the tree. Note: this can have ties sometimes (more than one with same right to be chosen, needs a tie breaker algorithm)

implementation: frontier is a FIFO-Queue, that is: nodes are taken from the front and are added to the end

problem: space consumption $1 + b^1 + b^2 + b^3 + \ldots + b^{d-1} + (b^d - 1)b \approx \mathcal{O}(b^{d+1})$ kills you quickly when d (depth of solution) is large.

BFS always finds a solution bcs it searches all depths, and goes to the next depth only after all nodes at the current depth are checked out.

if action costs (= costs for one edge in the tree) are 1, then BFS finds the cheapest solution. Otherwise not, bcs it finds the solution with the lowest depth! If path costs are unequal, then the cheapest solution can be very deep (but it has low costs along the path)

UCS

expand the node with the lowest cost of the path so far. Path cost is NOT the cost of the last action. it is the total cost of the path from start to the current state.

implementation: fringe is a queue ordered by path cost of nodes

UCS finds you guaranteed the cheapest solution - if action costs are lower bounded away from zero and positive.

Why ? 1. UCS explores all the cheaper nodes first. 2. if action costs are lower bounded and positive, then it takes only a finite number of steps to reach the cost level of the solution.

Lets explain this quickly:
at first a Counter example: solution has cost 5, but there is an infinite length path with ever decreasing costs $2^{-i}, i = 0, 1, 2, 3, \ldots$. following this path has a total cost of at most $\sum_{i=0}^{\infty} 2^{-i} = 2$. So we will never come at the point of expanding the node with cost 5. Instead we will traverse the path down to infinity. Thats why action

costs that are not lower bounded can be toxic! Their sum could form a converging total cost that can be very small :) .

Now if each action has cost if at least $\epsilon > 0$, then we know that all nodes at depth d_0 must all have a cost of at least $\epsilon \cdot d_0$ – without any expection. Certain depth – cost must reach a certain level.

So if, $\epsilon \cdot d_0$ is larger than the cost of the optimal solution C^*, then it means that the algorithm will never expand a node at the depth d_0. Simply because it will explore all cheaper solutions first, and run into the solution with cost $C^* < \epsilon d_0$

Problem: For action costs being constant UCS is same as BFS - and thus shares its memory problems!!

Note: the space complexity and time complexity formulas look so weird only for one reason: if each action has cost of at least $\epsilon > 0$, and the cheapest solution has cost C^*, then the solution must lie at depth at most $d = ceil(C^*/\epsilon)$.

So space and time complexity of $\mathcal{O}(b^{ceil(C^*/\epsilon)})$ is just because of that.

Technical note: In UCS you can hit a state multiple times. There might be multiple paths to Bukarest that pass the same city X in between!!

If you want to insert a node, that has a state that already exists in a node in the frontier, then compare the costs of the node already in the frontier and the node to be inserted. Keep the cheapest node.

DFS

Idea: expand the deepest node first, that is the node with the highest depth in the tree (ties...).

implementation: frontier is a stack, that is: nodes are taken from the front and are added to the front

Good side: space usage is linear in branch factor and max depth of the tree m – $\mathcal{O}(b \cdot m)$.

To see this: you expand always just one node at every depth, into b ones. so you maintain at every depth at most b nodes. b nodes times max depth of the tree m is as above.

problems: if the max depth of the tree is finite, then DFS may search all b^m nodes, even when solution lies at a shallow depth $d << m$. Even worse, in infinite spaces it may never find any solution at

all. Time complexity is not in terms of depth of the solution d but in terms of depth of the tree m - $\mathcal{O}(b^m)$!!

DFS is useful when there are many goal states in the tree (e.g. 30% of the nodes would have a state that meets the goal test!), and you do not care about the cost.

depth-limited search

depth-limited search is a DFS that stops when a depth l is reached. That is nodes in the tree with depth l are treated like actions do not change anything – terminal nodes. As a standalone algorithm its not clear what it should be good for. But ... That is useful for a hybrid search algorithm that we will see right below.

Time complexity is in terms of the depth limit l – $\mathcal{O}(b^l)$. Space complexity is similarly: $\mathcal{O}(b \cdot l)$.

Iterative Deepening Search (IDS)

Idea: run a for loop over depth limits.

$for\ l\ = 0, 1, 2, 3, \ldots$

- run depth-limited search until depth l
- terminate if a solution is found

Looks like a waste of time compared to BFS? Yes, but only moderately so!!

We assume the solution lies at depth d. What is the difference to BFS ($\mathcal{O}(b^d)$) ?

We know that iterative deepening will terminate at depth d. Until that you had expanded the top-level node d times. the nodes at depth 1 (there are b^1 many) were expanded $d-1$ times, the nodes at depth 2 (there are b^2 many) were expanded $d-2$ times, so total time complexity is:

$$db^0 + (d-1)b^1 + (d-2)b^2 + (d-3)b^3 + \ldots + b^d$$

This can be upper bounded by

$$\leq db^0 + db^1 + db^2 + db^3 + \ldots + db^d = d\sum_{i=0}^{d} b^k = dCb^{d+1} = \mathcal{O}(db^{d+1})$$

This is a moderate waste of space. But you have the memory efficiency of depth-first search - its linearity $\mathcal{O}(b \cdot l)$!

You trade space efficiency for time losses, no free lunch in algorithms.

Tree search vs Graph search

Tree search can expand a state repeatedly. Danger: infinite loops from this. But even with cycle pruning tree search will not always make you happy. Imagine the shortest path to Bukarest problem. There might be multiple paths from Arad to Bukarest that pass the same city *X* in between. There might be multiple paths from Arad to X but you want to find just the cheapest path from X to Bucharest.

Graph search maintains the set of already expanded/explored states to avoid such problems.

Idea is: keep an additional set of explored/visited/expanded states (not nodes!).

Whenever a node is taken off the fringe for expansion, then it is added to the explored/expanded set.

When expansion generates a new node, then check first:

- if a node with same state is in the frontier/fringe, keep the node with the lowest cost
- otherwise only if the new node is NOT in the expanded set, then add it in the fringe.

so expanded set acts as a guard to not add nodes to the fringe that were already explored!

Graph Search algorithm

the graph search in the slides is faulty (even though I took it from the ARMA instructors website). use this version please.

function graph-search(*problem*) **returns** a solution or failure
initnode ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST= 0
frontier ← your data frontier structure, *initnode* as its only element

frontier is a data structure, where pop() takes an element from the front
and where for BFS you insert new elements at the end
and where for DFS you insert new elements at the front
and where for UCS all elements are ordered when inserted by their path cost and cheapest comes first

explored ← empty set
loop do
  if frontier is empty :
    then return failure
  node=pop(*frontier*) # this chooses the lowest cost node

```
if true==problem.GOAL-TEST(node.STATE):
  then return(SOLUTION(node))
add node.STATE to explored
for each action in problem.ACTIONS(node.STATE) do:
  child ← CHILD-NODE(problem,node,action)
  if child.STATE is in frontier then
     keep in frontier that node of both that has lowest PATH-COST
    # if PATH-COST does not matter like in plain BFS or DFS, go to next else if instead
  else if child.STATE is not in explored or frontier then
     frontier ← INSERT(child,frontier)
```

Notes

You do not evaluate a state for the goal test, when you put a state on the fringe, but when you take a state of the fringe, UCS can show an example why this is necessary to find the cheapest path. suppose you have a direct edge in the tree from start to goal state, but it has very high cost. Doing the goal test when you put the goal state on the fringe would let you choose this very high cost path.

example: start state is S, goal state is G. doing a UCS with goal-test right when expanding a node (namely S), will let you choose the path with cost 100!