Uninformed Search

1 Breadth First Search

Uninformed Search (Blind Search) - Agents have no additional information beyond that provided in the definition of the search problem
Breadth First Search - Given the fringe of the search tree, every node on the fringe is expanded in each phase. I.e. a node with the smallest (search-tree) depth is expanded (in the case of a tie, order of expansion is an implementation detail)

Algorithm issues

- Need to create tree-nodes as required where tree-nodes need to be distinct but different tree-nodes might have the same associated state or action
- Need to retain enough information to reconstruct a path through the state space to a goal state, once we find one. Equivalently, a path from the root-node to a goal-node in the search tree
- We cannot be sure of the size of the state space or how much of the search tree we'll need to expand when we start
- We'll use specific data structures as and when they are needed

2 Data Structures of our search tree

Every node of our search tree has a unique identity, an integer id, and has associated with it the following data structure:

```
(id, state, parent_id, action, path_cost, depth)
```

where

- **id** is the unique integral identifier of the node (1 for root)
- **state** is the associated state from the state space (initial state for root)
- **parent_id** is the identifier of the parent of the node in the search tree (empty for root)
- **action** is the action which enabled the transition from the parent-node to the node in question (empty for root)
- **path_cost** is the cumulative cost of the step-costs in the path from the root of the search tree to the node in question (0 for root)
- **depth** is the depth of the node in question in the search tree (0 for root)

The transition function, the initial state, the goal test, and so on, all come as part of the search problem instance We reiterate:

- There could be two children with the same parent in the search tree, where the associated states of the children are identical but where the associated actions are different, and vice versa
- These children will have unique identifiers

3 Breadth-first search

```
newid = 1 #initialize node identifier
register node:
S[1]=initial_state; #Associate initial state with root
P[1] = -; #Root has no parent
A[1] = -; # No action to get to root
PC[1] = 0; #No path cost to get to root
D[1] = 0 #Depth of root is 0
```

```
initialize fifo queue (of tuples):
fringe F = [(newid, S[1], P[1], A[1], PC[1], D[1])]
if node 1 is a goal-node then return 1 #Halt when find goal node on fringe
else
  while F ≠ ∅ do
    pop (id, state, parent_id, action, path_cost, depth) from F
    #Child reachable in 1 step
    for each state child and action α for which we have that child ∈ϕ(state, α) do
      newid = newid + 1 #increment identifier
      register node:
      S[newid]= child; #associated state is child
      P[newid] = id; #parent is id
      A[newid] = α #action is α
      #Path cost is path-cost to parent id+ step-cost to new id
      PC[newid] = PC[id]+σ(state, α, child)
      D[newid]=D[id] + 1 #depth is depth to parent + 1
      if node newid is a goal-node then return newid
      # Put a non-goal node on fringe
      else add (newid, S[newid], P[newid], A[newid], A[newid], PC[newid], D[newid]) to F
return 0
```

Can we ever delete some of the search-tree from memory?

- Expand all nodes from root
- Keep expanding
- Can only remove from memory if there are no more nodes beneath it as you can't know if there would be a goal node beneath it, in which case, the node would form part of the path

Why do we include PC and D when we can reconstruct them?

- The function that determines if a goal is a goal node depends on them

4 Measuring performance

We evaluate a search strategy using four parameters

1. **Completeness** - Is the algorithm guaranteed to find a solution if there is one?
2. **Optimality** - If the strategy finds a solution, does it find an optimal solution?
3. **Time complexity** - How long does it take to find a solution?
4. **Space complexity** - How much memory does it take to perform the search

- Time/space complexity associated with specific notions of "size" or "difficulty" - we need these notions as our state space or search trees might be infinite
  - b: the **branching factor**, namely the maximum number of children of a tree node - the maximum number of successors of any state
  - d: the depth of the shallowest goal-node in the search tree - the smallest number of transitions from the initial state to a goal in the state space
  - m: the maximum length of a path from the root in the search tree - the maximum length of a walk in the state space
- The parameters b and m might be infinite
- d might not be defined

Time: The number of tree-nodes generated (or revealed)
Space: The number of tree-nodes held in memory at any one time

5 Performance of BFS

- Suppose that our state space is such that every state has exactly b successors i.e. the branching factor is b(and so is bounded)
- The root generates b nodes (at depth 1), and each of these b nodes generates b nodes (at depth 2)
- Each of these b^2 nodes (at depth 2) generates b nodes at depth 3, and so on
- Hence the number of nodes generated at depth at most d is

$$1 + b + b^2 + b^3 + ... + b^d$$

$$= (b^{d+1} - 1)/(b - 1) = \Omega(b^d)$$

- Note that all of these nodes need to be held in storage as potentially any one of them could appear on a path from some goal node to the root
- Thus, both the time and the space complexity is exponential in the shallowest depth d of a goal node. Equivalently, the nearest goal state to the initial state in the search space, in terms of the number of transitions
- BFS is complete (assuming bounded branching factor, if infinite, spend all time on first level of tree) but not necessarily optimal (might be shorter path cost at larger depth, which would then be more optimal)

6 Depth-First Search

Depth First search - Given the fringe of the search tree, a node of greatest depth is expanded

DFS is implemented by identical code to that implementing BFS, except that the queue is now a lifo queue (also called a stack)
DFS also has a (more well-known) recursive implementation

6.1 DFS Performance

- DFS is not complete as could go down a tree forever
- However, DFS requires considerably less memory than DFS. Once we ”back up” from a node, the register data corresponding to that node (and to all its ancestors) can be expunged from memory as we know that the node will not be involved in any solution
- This means only having to store all nodes on the path p from the root up to the current node and the other children of all nodes on path p
- Thus, the amount of memory required for a DFS is $O(bm)$, where b is the branching factor and m is the maximum depth of any node (not the shallowest depth of any goal node) in the search tree (if it exists). Even in such a search tree, DFS may take time exponential in m

7 Iterative deepening

- We can implement BFS using DFS
- Iterative deepening:
  - Undertake DFS but cut paths off at depth 1, if goal node found then return
  - Undertake DFS but cut paths off at depth 2, if goal node found then return
  - and so on
- Note that
  - We only ever keep a linear ($O(bh)$) number of nodes in memory but in order to check paths of length k+1 we have to recompute all paths of length k
- Complexity

- Complete (assuming bounded branching factor) but not necessarily optimal
- The number of nodes held in memory is $O(bd)$
- The number of nodes generated when goal is at depth d is

$$db + (d-1)b^2 + ... + 2b^{d-1} + b^d = \Omega(b^{d+1})$$

8 Assignment

- Implement 2 of the algorithms we cover in the course
- Solve the TSP
- Find the ”best tour” using the algorithms
- More marks for harder algorithms
- Tweaking the algorithms to give better results
- Also ran on secret city sets