Adam Rohdin - adaro815
David Tran - davtr766

Lab2 - Search

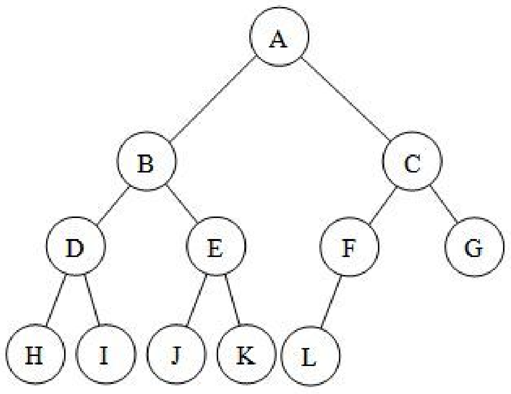
1. In the vacuum cleaner domain in part 1, what were the states and actions? What is the branching factor?
Answer: The states are the position of the nodes in the grid and the actions are the path from current position to the node. The branching factor is four because we expand in all directions from the current node.

2. What is the difference between Breadth First Search and Uniform Cost Search in a domain where the cost of each action is 1?
Answer: Uniform Cost Search is a search algorithm that does not involve any heuristics, instead it branches to the minimum cumulative cost path of a directed graph. However, if the cost of each action is 1 then there is no difference between the two search algorithms.

3. Suppose that h1 and h2 are admissible heuristics (used in for example A*). Which of the following are also admissible?
a) (h1+h2)/2
b) 2h1
c) max (h1,h2)
Answer: The first alternative is admissible because it guarantees that the heuristic will be between h1 and h2. The last alternative is also admissible because we are choosing the maximum value of two admissible heuristics which will guarantee that the estimated cost will be lower or equal to the actual cost. However, alternative two is non admissible because we cannot guarantee by multiplying an admissible heuristic will ensure that it will not exceed the actual cost.

4. If one would use A* to search for a path to one specific square in the vacuum domain, what could the heuristic (h) be? The cost function (g)? Is it an admissible heuristic?
Answer: An appropriate heuristic would be the manhattan distance as proposed in lab1 which is:
H = abs(currentPos.x - goalPos.x) + abs(currentPos.y - goalPos.y). The cost function *G* could be the cost to reach from current node to adjacent nodes. This would be an admissible heuristic.

5. Draw and explain. Choose your three favorite search algorithms and apply them to any problem domain (it might be a good idea to use a domain where you can identify a good heuristic function). Draw the search tree for them, and explain how they proceed in the searching. Also include the memory usage. You can attach a hand-made drawing.
Answer:
Given the following tree:



Adam Rohdin - adaro815
David Tran - davtr766

Depth-first search:
Depth-first search is one of two common strategy to search through a tree. DFS expands nodes that can be reached from current node, before continuing to the next adjacent nodes that have been stored in the frontier. The memory usage of the algorithm depends on the implementation because it can both be implemented recursively and iteratively. As we traverse down a tree we store only a single path from the root to a leaf node along with the unexpanded sibling nodes for each node on the path. When we have explored a node we can pop it from the stack and proceed to explore the sibling nodes. For a state space with branching factor b and maximum depth m, depth-first search requires storage of only O(b*m) nodes.
In the above tree, the nodes are processed in the following order:
[A,B,D,H,I,E,J,K,C,F,L,G].

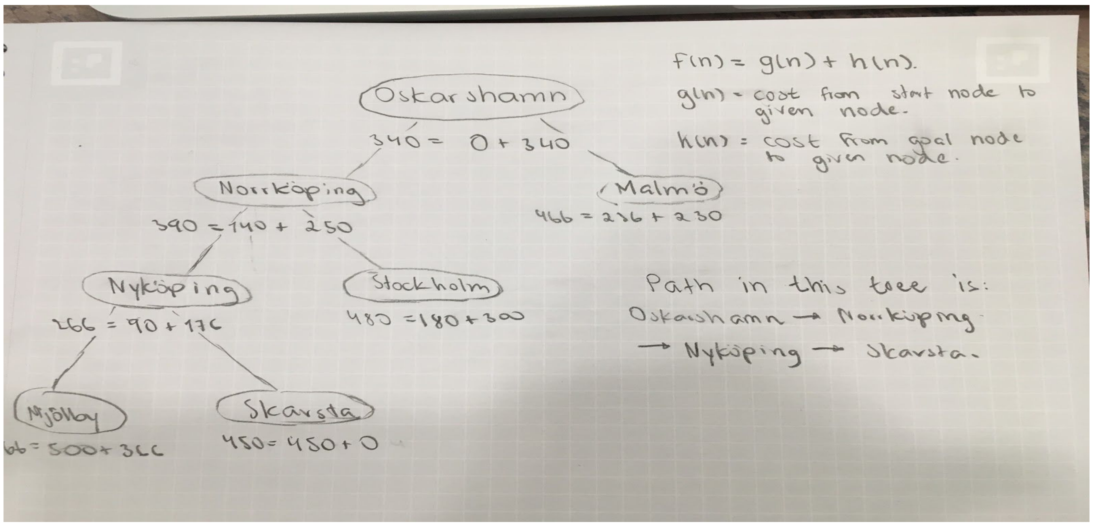
Breadth-first search:
Breadth-first search (BFS) is the second strategy to search through a tree. BFS visits current node and all adjacent nodes before moving to the next one. In terms of a tree structure, all nodes are expanded in a given level of the tree, before traversing down to the next level and so on. The space complexity of the algorithm is O(b^d) where b is the branching factor and d is the depth.
In the above tree, the nodes are processed in the following order:
[A,B,C,D,E,F,G,H,I,J,K,L].

For the last search, we propose the **A* search algorithm:**

A* search is one of the most popular search algorithms for graphs traversals and path-finding. This algorithm works by choosing the node with the lowest cost. The evaluation are made with this equation:

f(n) = g(n) + h(n)

Where *g(n)* determines the path cost from start node to given node and *h(n)* is the cost from the node to goal node. Therefore *f(n)* is the estimated cost of the cheapest solution through *n*. The space complexity of the algorithm is *O(b^d)* where *b* is the branching factor and *d* is the depth of the solution.



Adam Rohdin - adaro815
David Tran - davtr766

6. Look at all the offline search algorithms presented in chapter 3 plus A* search. Are they complete? Are they optimal? Explain why!

Answer:

Depth-first search is not complete nor is it optimal because if given infinite states there is a possibility that the goal state may never be found and it can be stuck in an infinite loop.

Breadth-first search is complete because it is guaranteed to find a goal state if it exists. However, the optimality of breadth-first search is highly depended on if the tree is weighted or unweighted.

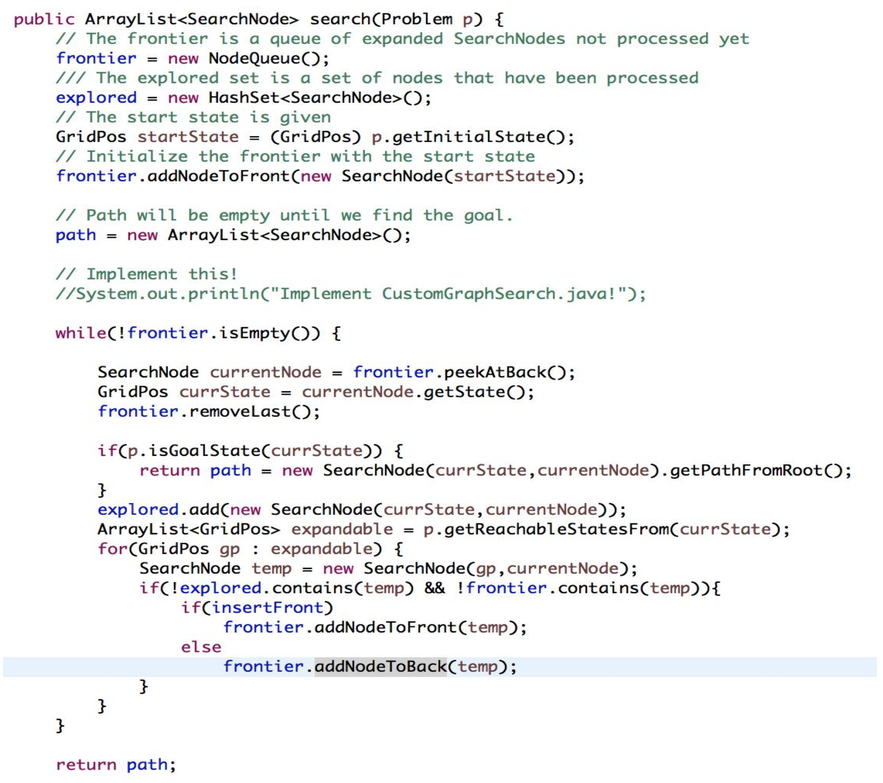
A*star conditions for optimality is that the heuristic *h(n)* must be admissible (tree-search version) and consistent(graph-search version). *h(n)* is consistent if the values of f(n) along any path are nondecreasing. This is to ensure that the algorithm never overestimate the cost. The completeness of the algorithm requires that there should be only finitely many nodes with cost less than or equal to the cost of the optimal solution path and the branching factor should be finite..

7. Assume that you had to go back and do Lab 1/Task 2 once more (if you did not use search already). Remember that the agent did not have perfect knowledge of the environment but had to explore it incrementally. Which of the search algorithms you have learned would be most suited in this situation to guide the agent's execution? What would you search for? Give an example.

Answer:

We did use A* search algorithm in lab1 which is explained in our lab 1 report.

Our implementation in lab2:



```
public ArrayList<SearchNode> search(Problem p) {
    // The frontier is a queue of expanded SearchNodes not processed yet
    frontier = new NodeQueue();
    /// The explored set is a set of nodes that have been processed
    explored = new HashSet<SearchNode>();
    // The start state is given
    GridPos startState = (GridPos) p.getInitialState();
    // Initialize the frontier with the start state
    frontier.addNodeToFront(new SearchNode(startState));

    // Path will be empty until we find the goal.
    path = new ArrayList<SearchNode>();

    // Implement this!
    //System.out.println("Implement CustomGraphSearch.java!");

    while(!frontier.isEmpty()) {

        SearchNode currentNode = frontier.peekAtBack();
        GridPos currState = currentNode.getState();
        frontier.removeLast();

        if(p.isGoalState(currState)) {
            return path = new SearchNode(currState,currentNode).getPathFromRoot();
        }
        explored.add(new SearchNode(currState,currentNode));
        ArrayList<GridPos> expandable = p.getReachableStatesFrom(currState);
        for(GridPos gp : expandable) {
            SearchNode temp = new SearchNode(gp,currentNode);
            if(!explored.contains(temp) && !frontier.contains(temp)){
                if(insertFront)
                    frontier.addNodeToFront(temp);
                else
                    frontier.addNodeToBack(temp);
            }
        }
    }

    return path;
```