Omar Mowafi 19-7322 T-12
Amir Emad 19-6642 T-14
Mohannad Banayosi 19-3702 T-13

A robot inhabits a rectangular grid of squares. The grid is surrounded by a barbed wire fence. The robot's body consists of multiple identical hardware units, each occupying a single square of the grid. The units do not necessarily occupy contiguous squares the robot's body parts are distributed all over the grid. The robot is controlled by a single station outside the grid. If two robotic parts occupy contiguous squares, then they are assembled into a single, larger part.

This is a normal search problem which contains two main subproblems : "Exploring" and "Search Technique".

1. “Exploring” : The exploring part of the problem is the part which is common and mutual in all search strategies (bfs, dfs, etc..). The task of exploring is when given a node (in our case the node represents a state and a state represents a state of the grid with parts and obstacles distributed) it expands it; and by expanding we mean looking for all possible states that can be reached from the node we got. This part of the problem is represented in the code by a method called *move(state current)* which takes as input a state and returns all possible “reachable” states from this state *current*.
2. "Search Technique" : The search method or technique is the thing that changes the order in which the nodes are explored depending on the technique. We have a method for each search technique.

To handle the parts and their movements, we keep track of the connected parts by checking it’s neighbour cells for parts. When trying to move the parts, we don't move one part alone, we move the entire parts that are connected.

Description of the main functions:

1. is_target(state)
   Method used to check whether the state is a goal state.

2. move (state)
   discovers the current state and returns an arraylist of states.

3. display(grid)
   prints the given 2D grid in an understable format

4. dispalynodes(arraylist)
   prints arraylist of the states visited by the search algorithm

5. bfs(state, char visualize)
   Uses bfs to search for a solution given the initial state of the grid

visualize prints the path of the search algorithm as it proceeds.

6. dfs(state, char visualize)
   Uses dfs to search for a solution given the initial state of the grid
   visualize prints the path of the search algorithm as it proceeds.

7. uniform(state, char visualize)
   Uses uniform to search for a solution given the initial state of the grid
   visualize prints the path of the search algorithm as it proceeds.

8. iterative(state, char visualize)
   Uses Iterative deepening to search for a solution given the initial state of the grid
   visualize prints the path of the search algorithm as it proceeds.

9. greedy(state, char visualize , String heuristic)
   Uses greedy based on a heuristic to search for a solution given the initial state of the grid
   visualize prints the path of the search algorithm as it proceeds.

9. aStar(state, char visualize , String heuristic)
   Uses aStar based on a heuristic to search for a solution given the initial state of the grid
   visualize prints the path of the search algorithm as it proceeds.

10. bestOptionGreedy(Arraylist)
    traverses on an arraylist of states and returns the state with the least heuristic. The returned state is the best option for the algorithm to take.

11. bestOptionAstar(Arraylist)
    traverses on an arraylist of states and returns the state with the least heuristic + cost. The returned state is the best option for the algorithm to take.

12. heuristic1(grid)
    Uses the grid and function, and the returns the best heuristic estimate at that grid shape.

13. heuristic2(grid)
    Uses the grid and function, and the returns the best heuristic estimate at that grid shape.

14. GenGrid()
    Generates a random grid.

15. Search(grid, algorithm, verbose)
    Aggregator of the search algortithms.

Search Algorithims :

1. BFS: For the bfs, we used the function *move(state)* that simply expands a given node. Using a queue (which has the First-In-First-Out characteristic) we were able to expand the nodes that are in the same level first before going to the next level.
2. DFS: For the dfs, we used the function *move(state)* that simply expands a given node. Using a stack (which has the First-In-Last-Out characteristic) we were able to expand the child of a node, then the child of this child node and so on until we reach a node that has no children, back track and expand another branch.
3. Uniform: The function with a normal arraylist that acts as the queue for expanded nodes but it only deques the node with the least cost regardless of its order and keeps doing the same thing until the queue is empty.
4. Iterative Deepening: The function works similar to DFS but with a new variable called curMaxLvl which it stops expanding nodes of the equivalent level since the maximum level is reached.This variable is incremented everytime it exits the loop.
5. Greedy: The function calls the heuristic function to calculate the heuristic value for the initial node. The function then checks whether it is a target state or not, if not then it checks whether the state is previously discovered or not. If it is then it continues. If not then it calls the function move and gets the state with the least heuristic value and continues the search.
6. A*: The function calls the heuristic function to calculate the heuristic + cost value for the initial node. The function then checks whether it is a target state or not, if not then it checks whether the state is previously discovered or not. If it is then it continues. If not then it calls the function move and gets the state with the least heuristic + cost value and continues the search.

Discussion of the heuristic functions:

Heuristic function one: (Manhattan distance)

The first heuristic function is known as the Manhattan distance. This heuristic function takes holds one robot part and sums the distances from all other robot parts. This heuristic is admissible because it never overshoots the cost to reach a goal state, because it returns the exact sum of distances from all the robot parts to a specific point.

heuristic function two: (diagonal distance)

This second heuristic attempts to get almost the straight line distance between the robot parts and one specific robot part. This heuristic is admissible because it attempts to calculate the straight line distance between the robot parts.