AI Assignment 1 Report

Loay Elzobaidy and Ahmed El Sayegh

October 18, 2017

1 Problem Description

In the problem we want to implement a searching agent. This agent can search for the goal state in this problem. The agent here is the R2D2 has two goals the first one is putting all the rock on a pad. It is preferable here to be the nearest pad so it consumes less steps. The second goal is to reach the teleporter cell.

2 Search Tree Node ADT

The search tree node is defined in our code as a class which contains 5 attributes:

- State: Contains the current state of this node
- Cost: Contains the cost of reaching this node from the root
- Depth: Contains the depth level of this node from the root
- Parent: Contains a reference to the parent node
- Operator: Contains the operator that, when applied to the parent node state, results in the current node state

We also use two functions $createNode$ and $nextState$. Both of them are helper functions. $nextState$ simply takes the current state and the operator and calls the function next in our TransitionFunction module. The function takes the state and the operator and returns a new state object after checking all the conditions.
$createNode$ is a wrapper function that creates a new Search Node.

3 Search Problem ADT

The search problem is defined in our code as a class which contains 5 attributes:

- initialState: Contains the initial state of this problem

- Operators: Contains a list of possible operators, which are up, down, left and right in our case
- goalTestFunction: Function that takes a state and returns true if it is a goal state
- pathCostFunction: Given a node, it returns the cost of reaching the given state from the root

We pass all these attributes to the problem in the index.py file, which can be considered our HelpR2-D2 subclass which will be discussed shortly

4 HelpR2-D2 Subclass

Our HelpR2-D2 Subclass can be considered to be the index.py file. In this file, we initialize and pass all parameters to define our search problem and our grid. We first initialize our grid, then fill it up with rocks and place a teleporter and R2D2, we also define m as a global variable, which has a default value of 5. We also define the operators and the goal test functions and the path cost functions. We then create our search problem and pass that to our generic search function. We iterate over all the possible queuing functions and solve the problem six times, once with every function. We will discuss the results of the different queuing functions shortly. After the search is finished and results are found, we process the results into a dictionary and pass that dictionary to the GUI component of the project, which handles showing the different results found by different queuing functions, featuring step by step moving as shown in Figure 1.

5 Main Functions problems

The implementation is divided into eleven different files. The first file is the Generic search file, which is responsible for implementing the search technique mean function. First Search function, this is the generic search function. This function takes as input problem, strategy, visualize function, and gui flag by default set to None. First it initialize a queue and keep dequeuing from the queue node by node. The node search stop when a goal state is reached.

A function that is called in the generic search function is the Queuing function, which is responsible for implementing all the search strategies takes as parameter the newNodes to be added to the queue, strategy for the search and the problem itself. The second file is the GridGenerator it is responsible for generating rock places which takes the value 1, placing pads takes as input -1, place the teleporter valued -2, and finally the R2D2 agent takes as value 2. and finally the place item function responsible for randomly placing the items on the grid.

The GUI file this where the visualization window is implemented. It uses Qt library in python to generate a window and refresh every time with the out.png image. The next file is the heuristic file, where the two heuristics are implemented. First there is two functions getDistance, which gets the euclidean distance between two points and getActualDistance function which gets the actual number of steps between two heuristics. We have the first heuristic function takes a grid an input which represent the state here and the same applies for heuristicTwo.

The index file responsible for running the program has mainly two functions output, run functions. This function is responsible for printing the result in the terminal. It takes as input the result node and the strategy. This function actually is making backtracking till it reaches the root. The run function is responsible for running the search problem code takes as input visualize . First it generates a grid and places the R2D2 with initialize some positions. initialize the goal state function and the past cost function. It loops over all the strategies and run the generic search over this grid. For implementing the Iterative deepening, we use a loop until infinity.

The next file is SearchNode, expand function creates for the current state, the children nodes. takes operators as input, for every action it creates the corresponding child node. The SearchProblem file, generate a class of search problem with the corresponding instances. The State file generate a State file, with five instances. The visualizer class, the first function is rendercell it renders cell by cell. The refresh function to refresh the frame.

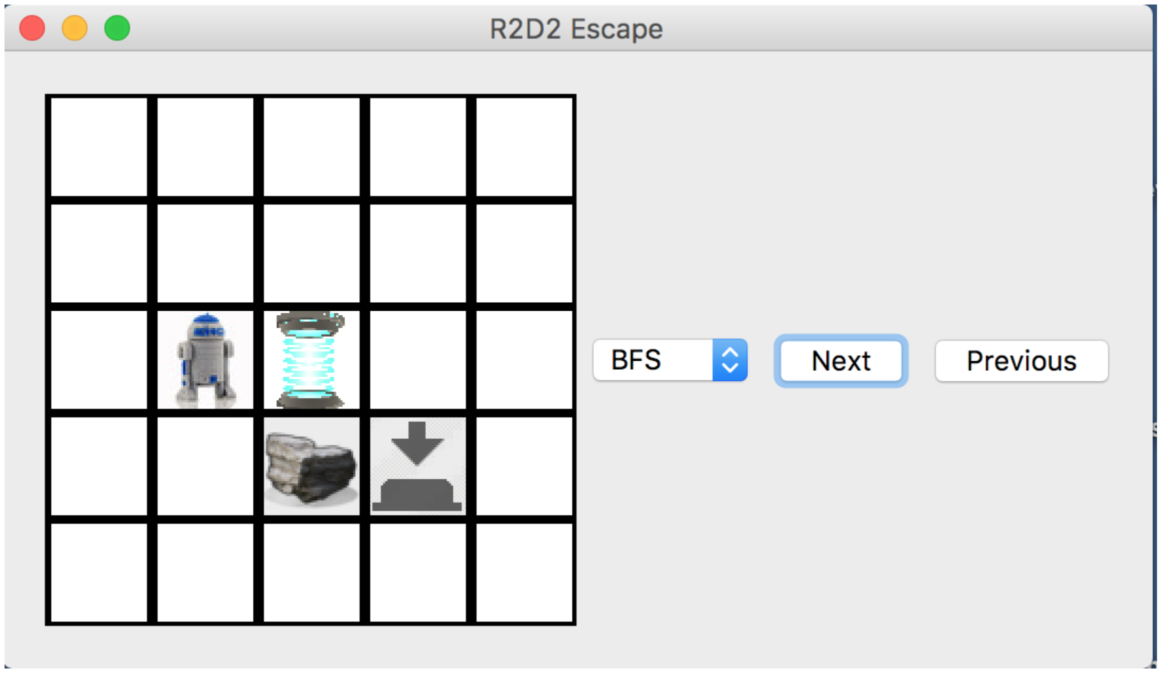


Figure 1: GUI Screenshot

The render grid function takes the grid and render it the images of the cell, cell, the R2D2, pad and teleporter images.

Tracking file, responsible for memo rising the states. It takes as input the next state, visitedstates check if the next state exist in the visited states. The TransitionFunction file contains only one function next. This function takes as input previous state and the operator and then check for all the operators if operator up for example, checks if the R2D2 is on the edges. if there is a rock upper to it it shfts the rock else it moves to the upper row. It then replicates this to all the other operators in the search problem.

6 Various algorithms

In the generic search, we add the children nodes into a queue called NewNodes. we call the queuing function. The queueing function is responsible for implementing all the searching algorithms. First BFS Which concatenates the NewNodes queue to the end of the global queue. In the DFS it concatenates the old nodes queue in the NewNodes queue and then empty this queue in the global queue. The next strategy is the Uniform cost, it adds all the nodes in the list from the global queue and the new nodes queue. It sorts this list based on the depth in each node. and then adds all this nodes back to the queue. Iterative deepening is the same as DFS but only adds in the global queue if the nodes depth less than or equal to the maximum depth so far. The greedy approach initialize two lists the cost and the nodelist lists puts all the nodes in the nodelist and then append in the cost list all the weights for each node based on the heuristics. After that calling the numpy.argsort which returns an array with each weight position in the sorted array without sorting it. After that we insert with this order the corresponding node.Finally we insert all this nodes in the global queue. In the A* algorithm it the same as the previous one, except that we ordered based on the heuristic function added to the depth.

7 Comparison

For the same search problem we ran all the algorithms and got some statistics The BFS approach took 71 expanded nodes, got a path of ['Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Left', 'Down', 'Down'], which is here the optimal path, it is complete because it will find a solution if there is a one.

The DFS approach took 27 expanded nodes, got a path of ['Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Up', 'Left', 'Left', 'Down', 'Down', 'Right', 'Right'], which is not the optimal path, it is not complete because it can expand an infinite branch but here we made it complete because we memorize the state which is represented by a grid and the R2D2 does not go to a visited state again. By doing this it is complete but it is not optimal.

The UC approach took 71 expanded nodes, got a path of ['Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Left', 'Down', 'Down'], which is the optimal path, it is complete because if will definitely find a solution if there is one at the worst case scenario the Uniform coast search algorithm will expand the whole nodes.

The Greedy algorithm took 144 expanded nodes, got a path of ['Up', 'Left', 'Left', 'Down', 'Right', 'Left', 'Down', 'Down', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up'] which is not the optimal path, it is complete because in the worst case scenario it will expand the whole nodes.

The A* algorithm took 63 expanded nodes, got a path of['Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Left', 'Down', 'Down'] which is optimal and complete because if there is a solution it will reach it eventually. it took less number of nodes than the BFS and the UC.

The Iterative deepening, it took 277 expanded nodes, got a path of ['Up', 'Left', 'Left', 'Down', 'Right', 'Up', 'Right', 'Down', 'Left', 'Down', 'Down'] which is the optimal path, it is complete because it will reach a solution if there is one.

8 Admissibility of heuristic functions

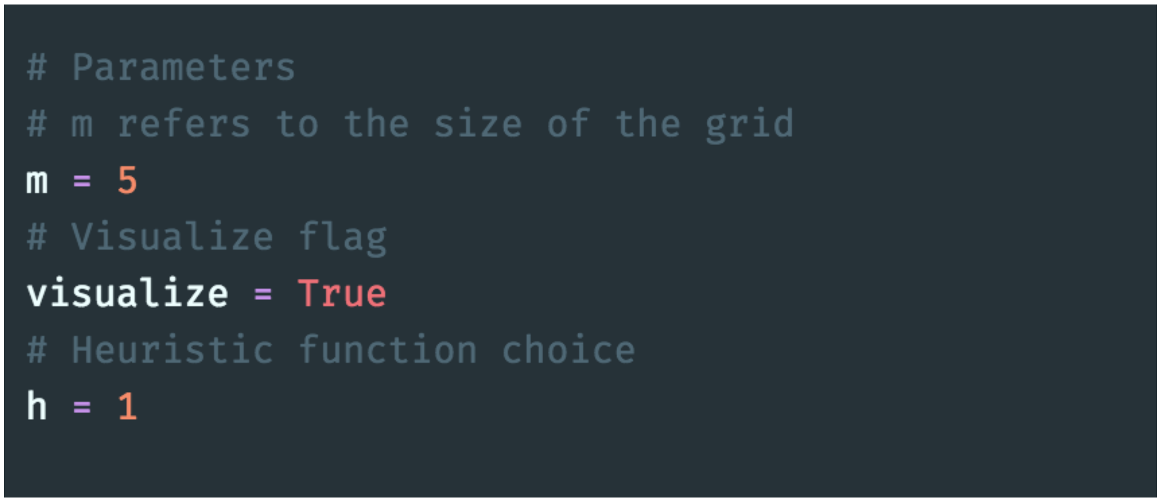
We came up with two heuristic functions. The first one calculates the euclidean distance between R2D2 and the nearest rock, then adds to that the euclidean distance between the aforementioned rock and the rock nearest to it. We keep finding the closest rocks and adding euclidean distances until all rocks have been passed by once, then we calculate the euclidean distance between the last rock and the teleporter. This heuristic function is admissible because it uses the euclidean distance, so the distance is, in fact, smaller than the actual number of steps R2D2 needs to take in order to reach any of the rocks. It is also admissible because we do not add distances between rocks to pads, meaning that even if every rock was one step away from a pad, the heuristic function would still estimate a cost under the actual cost since.

The second heuristic function is very similar to the first one, with one difference. We calculate the actual distance instead of the euclidean distance. Like the first heuristic, this function is admissible because it does not count the distance required to get the rocks onto the pads, even if that distance is just one step.

9 List of instructions

First, you need to install python3 on your machine and then run command `pip3 install -r requirements.txt`. Then to run the project you need to run this command `python3 index.py`. In index.py, we extracted the parameters into one part of the file, as shown in

Figure 2 variable m represents the grid height and width. The h variable can be either 1 or 2 representing either the first or the second heuristic. If the visualize flag is true, then, the GUI will be displayed after all the paths are found. The GUI can be used to select any approach and move forwards and backwards step by step in the solution as shown in Figure 1



```python
# Parameters
# m refers to the size of the grid
m = 5
# Visualize flag
visualize = True
# Heuristic function choice
h = 1
```

Figure 2: Parameters in index.py