Introduction

This section will have a **brief description of the problem**. The R2-D2 is a robot that is trying to escape from an m x n grid. The problem is that the robot’s only way out is a teleport laying on one of the cells of the grid which can send it back to its planet. Yet, it is not as easy as just finding that cell and use it to get back. The grid also consists of three more things; pressure pads, obstacles and rocks.

For the robot to use the teleport cell, he must push a rock over each pressure pad first. Still not an easy task giving that the robot has some limitations. The robot cannot push more than one rock at a time in any given direction neither push any of the prison’s obstacles. After all the pressure pads have a rock over them, the teleport cell is finally activated, and the robot can head directly to it, use it and go back to his home planet.

Implementation

Here, this section will go briefly through the **implementation and classes of the project**. To make use of the **OOP concepts**, Java is used to implement this project. The project consists of 5 packages:

1. Search Package:

This package contains all the general and abstract classes to solve any general problem. It has 5 classes:

### State Abstract Class:

The State object denotes the state of a given Node. Since each problem defines its own state, the State class is an empty Abstract class that has only one abstract method which is the *heuristic(int n) method.* This method will be discussed later.

### Node Class:

It defines a Node object of a general search tree. The Node object has a 5-tuple instance variable; parent of type Node which indicates the parent of this node, currentState of type State which denotes the state of the node, depth of type int indicating the level of the node in each search tree, pathCost of type int which is the summation of the nodes’ costs till reaching that node and operator of type String indicating the operator that was made by the parent node to reach that node. Then there is one final variable which is the order of type int, it is used to order the node in the priority queue in case of greedy or A\* algorithms. The search node tree is implemented as 5 tuple nodes as previously explained following Russell and Norvig. This node has a link to its parent and later used to build the search-tree

### Problem Abstract Class:

The Problem Class defines the 5 tuples of any general search problem. It has three instance variables; an array of type String operators which consists of all the possible operators of the problem, an array of type State stateSpace which is the finite state space of the problem and finally the initState of type State which defines the first state of the problem. Then there are the two missing items of the 5 tuple which are two abstract methods, the *goalTest(Node node)* which returns a Boolean of whether the goal test is passed or not and the *pathCost(Node n)* which defines the pathCost of a given node.

Finally, There are three important abstract methods; the Expand(Node node) which returns an ArrayList of type Node containing all the reachable nodes from the given node, pastState(Node node) which returns a Boolean denoting whether the given node has a state that has been explored before or not, this method enhances the search performance and speed by eliminating redundant states and lastly the clearPastState() which clears the ArrayList holding the past states for a purpose that would be explained later on in this report.

### QueuingFunction Enum:

Enumerators that differentiate between the different search algorithms.

### GeneralSearch Class:

This class holds most of the logic in this project. The class is defined by four instance variables; the problem of type Problem, the qingFunc which is an enum of type QueuingFunction stating which queuing function will be used in the search and two data structures. The reason two data structures are used in the search is that different search algorithms need different data structures, the algorithms are divided into two sets. BFS, DFS and IDS use a double ended queue which eases the process of adding the nodes in either side depending on the algorithm. UC, A\* and Greedy algorithms use a priority queue which also eases the process of sorting the nodes inside the queue. Then there is the search() method which either returns the node that passed the goal test or null indicating that no solution is found.

Depending on the qingFunc, the search() calls one of the six different search methods to handle the tree traversing and returns the solution. Later a deeper digging into how each algorithm is implemented will be stated. Finally, the main method of the class and the project is the search(Grid grid, QueuingFunction strategy, boolean visualization) method; as shown, the method takes a grid object of type Grid which defines the initial grid of the problem, a queuing function, and visualization of type Boolean which indicates of the user wants to visualize the problem or not.

1. Assignment1 Package:

This package contains the specifics for the R2D2 problem. It utilizes the Search package to be problem specific.

### MyState Class:

This class extends the State class and defines the HelpR2-D2 problem’s state. The state here keeps track of the things that usually change with the problem’s operations. It has currentPosition of type Cell which denotes the position of the droid, unactivatedPads denoting the number of the pads that still does not have a rock above them, an array of type Cell rocksPositions that holds the positions of rocks in the grid in each state and finally an int expandedNodes which records the number of nodes expanded to reach this node. The class also implements the heuristic method of the State abstract class. The heuristic(int n) takes an n as an input denoting which heuristic function will be used since there are two functions, 0 indicates using the first function and 1 indicates using the second. Both will be explained.

### HelpR2D2 class:

The class extends the Problem class. The HelpR2D2 class holds the instances that does not change with the change of the state, in other words, those which are static and problem specific. It takes a grid of Type Grid and initializes those instances from that grid. It has telePosition of type Cell which holds the teleport position cell, two Cell array s obstaclesPositions and padsPositions that defines both obstacles cells and pads cells respectively, the height and width of the grid, the numberOfExpandedNodes which keeps track of how many nodes have been expanded so far and finally an array list expandedStates of type MyState that keeps track of the already expanded states so as not to go through the same state twice in some of the algorithms.

The main method in the class is the *Expand(Node node)*, the method takes the node to be expanded and calls four methods for four operations on that node; up(Node node), down(Node node), right(Node node) and left(Node node). Each method checks if the operation can be applied on that node, if so it will apply the operation and returns the result node after that operation. The four methods check all the possible cases when the droid tries to move in any of the four directions. All the checks are well documented inside the source code.

3. Grid Package

The grid package includes the classes that represent the physical components of the grid. These physical components are the Cell class which represents a single cell in the grid, the grid class with includes the grid of cells and the enumeration of the cell status (whether it is an obstacle, pressure pad, teleport or a free cell).

### The Cell Class:

The cell class describes the cell. It includes the horizontal position (x), the vertical position (y), the cell name, a boolean that represents whether it has a rock on top of it and an enum that represents its status.

CellStatus Enum:

The status was implemented as an enum because it independent of whether it has a rock or not. An obstacle cell is the only status that does not allow to have a rock.

### The Grid Class:

The grid class represents a grid as data structure independent of the search algorithm, it does not have any search strategies implemented. It contains a 2D array of cells and two integers representing the width and the height of the grid. A grid object can be initialized either randomly or with specific width and height.

The obstacle and pad cells in the grid are randomly distributed in the grid as well as the rocks such that the number of pads equals to the number of rocks and the both the pad and obstacle counts is less than the number of cells divided by 7. Such constraints are implemented to provide an estimation of a grid that is realistic for the agent to traverse.

4. GUI Package

The GUI package adds an interactive GUI to visualize the solution in a clearer way.

### The Cell Class:

The cell class displays the different variations of the Cell, whether this cell contains a rock, obstacle, pad, teleport or agent each in different color

### The MainWindow Class:

This class is responsible of converting an object grid to GUI.Cell and drawing this grid to the user. Moreover, it enables the user to select between different algorithms the preview the motion of the robot at every step.

5. Test Package

This package is used to run the project

### The NoSolutionException Class:

This is an exception to indicate that the generated grid does not have a solution and enables us to automatically create a new Grid.

### The Test Class:

This class runs the project by first creating a grid through the method genGrid() then passes this grid to method search() to search for a solution in the grid.

### The GUITest Class:

This class is responsible of converting an object grid to GUI.Cell and drawing this grid to the user. Moreover, it enables the user to select between different algorithms the preview the motion of the robot at every step.

Discussion

This section will give a briefer explanation of the implementation. The section will discuss the main points required in the report. The previous section discussed the implementation of the search-tree node ADT, the implementation of the search problem ADT and the implementation of the HelpR2-D2 problem.

The Search Algorithm and main functions:

The GeneralSearch.search method checks the queuing function and calls a subroutine for each queuing function. BFS, DFS and IDS use a double ended queue to execute the search on the other hand, UC, Greedy and A\* use a priority queue. Each subroutine follows the generic search algorithm presented in Lecture 2 slide 11.

The tricky part of the algorithm is the expand function which is handled by an instance of the Problem Class which is a parameter of the GeneralSearch class. In this project, an instance of HelpR2D2 is passed to the GeneralSearch class which expands a node to a list of nodes in the semantics of the specific problem of R2D2.

In the HelpR2D2 class, a node is expanded based on the given operators (UP, DOWN, LEFT, RIGHT) with respect to the given grid. The expand function checks all the cases that allows of disallows R2D2 to go from a cell to an adjacent cell with respect to some factors like the obstacles, borders and rock positions. Such factors are extracted from the grid in the form of arrays of Cell objects for the obstacles and pads and rocks, and single Cell object for each the Agent position and the teleport position. The HelpR2D2 class is equipped with a goalTest function that indicates whether a given node is a goal node or not. A node is a goal node if the specific state MyState contains an agent standing on the teleport position and the number of unactivated pad is 0. A pad is said to be unactivated if there is no rock on top of it.

Moreover, there is the pastState(Node n) check which is checks if this node is a repeated state or not. To Add there is also the Grid.genGrid() function which generates a random grid with a size m x n where m,n can be any number between 3 to 6. Then it generates a random number of obstacles, pads and rocks and assigns them some random positions. Finally, there is the visualize method which is used to convert a certain state into a visualized grid on the console.

The Various Search Algorithms:

The search algorithm will traverse the tree differently depending on the queuing function that it takes. All the six algorithms are implemented. In the search algorithm the initial node is the first node that gets popped from the queue. The node is then expanded using the expand method inside the HelpR2D2 class. The expand method returns an array of possible child nodes by applying all the possible operators on the given node. The way those nodes are inserted inside the queue depends on the search algorithm that will be used which defines how the tree will be traversed.

BFS:

In Breadth First Search the tree is traversed level by level. So, we always enqueue the returned nodes from the expand method at the end of the queue so that the parents’ nodes are always explored first.

DFS:

In Depth First Search, it is quite the opposite from BFS. The tree is traversed branch by branch. If a branch ends, the algorithm starts backtracking. Thus, the returned nodes from the expand method are added at the beginning of the queue.

IDS:

In Iterative Deepening, a limited depth first algorithm is run repeatedly and every time the depth is incremented by one till a solution is found. To do this, we implemented two methods; *IDS(Deque<Node> nodes, Node intialNode)* and *IDS(Deque<Node> nodes, Node node, int depth).* The first one takes the nodes queue and the initial node and keeps calling the second method inside a loop till a solution is found. The second method here is the limited depth first search algorithm. Each time this method runs till the depth becomes less than the child nodes depth.

UC:

When the child nodes are returned from expanding the parent node, the nodes are inserted inside the queue and then the queue is sorted by the path cost. In this problem, since all operations yields 1 as a cost, the uniform cost works the same as the breadth first search.

Greedy:

In a greedy algorithm, all the child nodes returned from expanding the parent node are instead in the queue and then sorted by their heuristic function. The algorithm takes an int n; when set to 0 it denotes to the first heuristic function and when set to 1 it denotes to the second one.

A\*:

A\* algorithm is very similar to the Greedy one, except that the nodes are sorted by the heuristic function added to the path cost of the node. This insures finding the optimal solution which was not the case in Greedy

The Heuristic Functions And Their Admissibility:

The project has 2 heuristic values both returned by calling heuristic(int n) and if n is 0 it will return the first heuristic value and if n is any other number the second heuristic value will be returned. Both heuristic values are calculated in O(1); this simple calculation allows is very efficient as it adds no extra runtime to the search method used leaving it to run fast and efficient and in turn this saves a lot of runtime. Despite using simple functions but by trying out more complex functions these simple algorithm were found to be more efficient in experiments.

The first heuristic value is the number of unactivated pressure pads. At each state of the agent motion the function checks the number of unactivated pressure pads and uses this as the heuristic value for this current state. R2D2 has always to activate all pads before trying to escape. Therefore, every time he activates a pad the heuristic value will decrease indicating that he is close to the target. Moreover, since the R2D2 cannot move 2 stones in one move and cannot remove a stone that is on a pressure pad this insures that the heuristic value is always an underestimate for the real cost to escape if every step costs 1. Since this heuristic value is always an under estimate therefore it is admissible. To add it is also monotonic as at every step this function returns an underestimate for the real cost for the robot to reach the next state till his goal.

The second heuristic value is calculating the distance between the robot and the teleport position by getting the difference in X position minus the difference in Y position. R2D2 always must stand on top of the teleport to escape from the maze. Therefore, calculating the number of steps between R2D2 and the teleport is either exact or underestimate for the real cost needed to reach the target given that every step has a cost of 1. Since this heuristic value is always an under estimate therefore it is admissible. To add it is also monotonic as at every step this function returns an underestimate for the real cost for the robot to reach the next state till his goal.

Therefore, both heuristic functions are admissible and as simple as they are they have reached a better performance results than more complex values.

Comparing the Search Algorithms:

Before comparing the algorithm with samples this part will discuss the algorithms theoretically. The performance of the diﬀerent algorithms implemented will be compared in terms of completeness, optimality, and the number of expanded nodes.

Starting by completeness, generally only BFS, ID, Uniform Cost and A\* are complete and are assured to find a solution if there is one. On the other hand, greedy and DFS are not complete. But since repeated states is handed in this problem therefore greedy and DFS will also be complete as they cannot have stuck in the same loop for every and will expand only new states; since the state graph is always finite then the greedy and DFS will terminate at one point and find a solution.

Secondly is optimality, only Uniform Cost and A\* are generally assured to return the optimal solution. But since this problem insures that the cost between every node and its child is always 1 then BFS and ID are also optimal in this problem only.

Finally expanded nodes. When it comes to expanded nodes, it cannot be predicted between all six algorithms. Only A\* is assured to expand less nodes than the Uniform Cost algorithm, other than this case the number of expanded nodes depends on the states and cannot be predicted. Also to add, since this problem has a cost of 1 then the BFS is treated same as Uniform Cost this leads to both expanding same number of nodes.

Results and How to run

In this section will compare the results of a random grid and returning the results for this grid. Then it will give general instructions on how to run the project.

Results:

This will have a copy of the console output for a run of the program

WELCOME

Escape mission of R2D2 from the Death Star

========

=> Width = 3

=> Height = 5

=> Generating grid..

Initialization Log:

=> Initializing agent at R2 C2

=> Initializing teleport position at R1 C4

=> Initializing with 2 obstacles

=> Initializing with 2 pads and rocks

=> Obstacle at: R1 C3

=> Pad at: R2 C1

=> Pad at: R1 C2

=> Rock at: R2 C3

=> Rock at: R1 C1

========

The Grid:

\_ \_ \_

| | | |

\_ \_ \_

| | r | p |

\_ \_ \_

| | p | A |

\_ \_ \_

| | o | r |

\_ \_ \_

| | T | |

\_ \_ \_

SUCCESS!!

Using Expansion Strategy: DF

Solution Depth: 27

Expanded Nodes: 977

LEFT->LEFT->DOWN->DOWN->RIGHT->RIGHT->UP->DOWN->LEFT->LEFT->UP->UP->UP->UP->RIGHT->DOWN->LEFT->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

SUCCESS!!

Using Expansion Strategy: BF

Solution Depth: 15

Expanded Nodes: 313

UP->UP->LEFT->DOWN->LEFT->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

SUCCESS!!

Using Expansion Strategy: ID

Solution Depth: 15

Expanded Nodes: 4175

UP->UP->LEFT->DOWN->LEFT->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

SUCCESS!!

Using Expansion Strategy: GR1

Solution Depth: 17

Expanded Nodes: 412

UP->UP->LEFT->DOWN->UP->LEFT->DOWN->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

SUCCESS!!

Using Expansion Strategy: AS1

Solution Depth: 15

Expanded Nodes: 296

UP->UP->LEFT->DOWN->LEFT->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

SUCCESS!!

Using Expansion Strategy: GR2

Solution Depth: 15

Expanded Nodes: 345

UP->UP->LEFT->DOWN->LEFT->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

SUCCESS!!

Using Expansion Strategy: AS2

Solution Depth: 15

Expanded Nodes: 302

UP->UP->LEFT->DOWN->LEFT->DOWN->DOWN->DOWN->RIGHT->RIGHT->UP->UP->DOWN->DOWN->LEFT

As shown in the results all algorithms found a solution with AS1 expanding less nodes than any other algorithm and returning an optimal solution

The algorithm is executed by running a test file in the Test package. This test files instantiates a Grid object and calling the method GeneralSearch.search. The Grid object is passed to the method object along with a QueuingFunction enumeration and a boolean that indicates whether it is required to visualize the agent steps after finding a solution or not.