• A brief description of your understanding of the problem.

The problem is about a robot in a 2D grid which has pressure pads, rocks, obstacles and teleportal. The solution of the problem is when the robot push a rock over every pressure pad then reach the teleportal. The aim of the project is to give an answer if given this grid with the position of things listed above, will there be any way to complete this process or not, and If a solution exists show it. The implementation was done using java.

• A discussion of your implementation of the search-tree node ADT.

Classes explained: Grid, Node, State, R2D2State

The search-tree node (Node class) had 3 instances: the node’s State, the path cost of this node, and the parent node. The state will be explained below, the path cost can also be called the depth of the node, and the parent node is used to store the path the robot used if there exists a solution. There is an abstract class state and a subclass called R2D2State which extends the class State. R2D2State has 3 instances an object Grid which carries the information about the grid in the current state (will be explained in more details below), the number of pressure pads left with no rock on them, and the previous operation done that lead us to this state. It also contains the 2 heuristic functions (explained below). As for the Grid class, it has 6 instances: the number of rows and columns, the actual grid (2D array), a boolean 2D array called visited which denotes the visited cells since the last time a rock was moved, the current position of R2D2 and the position of the teleportal.

• A discussion of your implementation of the search problem ADT. • A discussion of your implementation of the HelpR2-D2 problem.

Classes explained: SearchProblem, HelpR2D2

An abstract class called SearchProblem was created it has 2 instances: Initial state and a set of operators. A subclass HelpR2D2 that extends SearchProblem was created.

The class contains GenGrid() method that randomly generates an initial grid, a constructor, it has the method goalTest(Node n) which returns true if the node satisfies the goal test, it has the method pathCost(Node n), also the method ArrayList<Node> expand (Node n) which takes a node and returns an array list of its children nodes based on the operators available. A method valid and its helper method, which check if R2D2 can move to a certain cell. A method update and its two helper methods, which generate an updated 2D grid representing the new state, based on the direction of motion (operation) and the current state (considers the rocks moved, pressure pads activated, etc.)

• A description of the main functions you implemented.

The following are functions in the HelpR2D2 class.

- GenGrid: generates a 2D character array with dimentions 3x3 (due to memory and time limitations on larger dimensions), places rocks, pressure pads, obstacles, a teleportal, and R2D2 randomly on this grid (without the possibility of R2D2 being placed on a pressure pad or teleportal, even though it is handled during searching). And creates a Grid object with the relevant information, including a 2D boolean array visited (as described before).

Here is a list of characters and their meaning regarding the 2D grid:

'-' means a blank space, 'I' means the current location of R2D2 if it is standing on a blank space, 'R' means there is a rock at that position, 'P' means there is a Pressure pad at that position, If R2D2 is standing on a pressure pad it becomes 'B', 'O' means there is an obstacle at that position or there is a rock on a pressure pad which was turned to an obstacle, 'T' means the teleportal position, If R2D2 is standing on a Teleportal it becomes 'E', If a rock is pushed on a Teleportal it becomes 'W’.

**-boolean** validHelper(**int** i, **int** j, Grid g): (i,j) is the position being checked, it checks if this position is not out of bound, nor an obstacle.

-**boolean** valid(**int** i, **int** j, Grid g, **char** d): d indicates the direction r2d2 is moving. The method checks if R2D2 can move into this position (knowing that it came here using the transition d) i.e. if it’s a blank space/pressure pad/teleportal, or a rock that can be moved in the direction d.

-**int** update(**char**[][] grid, **int** oldi, **int** oldj, **int** newi, **int** newj): updates the grid and returns 0 if no rocks were moved, 1 if a rock was moved, 2 if that rock was moved onto a pressure pad. It updates R2D2’s new position and calls *updateOldPosition. It also calls* *updateRockNewPosition if R2D2’s new position contained a rock, to update where that rock would be pushed.*

-**void** updateOldPosition(**char**[][] grid, **int** oldi, **int** oldj) updates this position (by moving R2D2 out of it & updating the character accordingly)

-**boolean** updateRockNewPosition(**char**[][] grid, **int** rocki, **int** rockj) updates this position (by moving a rock into it & updating the character accordingly) returns false if no pressure pads were activated, true if one was.

-ArrayList<Node> expand(Node n): checks the available operators, and tries to expand into each corresponding cell (R,L,U,D), it checks if moving into this cell is valid (explained above), if so, it creates a new node with the relevant information (new state(with an updated grid), path cost, and the parent of this node (the current node being expanded)). The visited array is reset if R2D2 moved a rock, otherwise it is the same as the old node. And in all cases R2D2’s new position is considered visited.

• A discussion of how you implemented the various search algorithms.

Classes explained: GenericSearch

- **void** Search( Grid grid, String strategy, **boolean** visualize) it searches for a solution for the search problem (HelpR2D2) with the Grid grid as it’s initial state, and uses the strategy for searching. The method prints the original grid, and then prints the sequence of steps (the first step at the bottom) used to reach the solution (if one was discovered). When visualize is set to true, it also prints a visual presentation of the grid as it undergoes the different steps of the discovered solution (if one was discovered), otherwise it prints NO SOLUTION along with the original grid.

In the case that the strategy is Iterative Deepening(ID), a loop is created which initializes a variable depth to 0, and keeps incrementing it while calling depthLimitSearch (explained later) with the problem and the new depth, until a solution is found (if any).

-Node genericSearchAlgorithm(SearchProblem p, String QingFun): It creates an initial node from the search problem’s initial state, adds it to an arraylist of nodes. It searches by removing the first node of the array, applying goal test to the node and returning it if it satisfies the test, otherwise it expands the node and inserts the expanded nodes into the arraylist based on the search strategy’s Queueing function. In the case of:

Depth First (DF): expanded nodes are added at the beginning of the list.

Breadth First (BF): expanded nodes are added to the end of the list.

Uniform Cost (UC):expanded nodes are added to the list, then the list is sorted based on the path cost of each node.

Greedy (GRx): expanded nodes are added to the list, then the list is sorted based on the heuristic function x.

A\* (ASx): expanded nodes are added to the list, then the list is sorted based on the heuristic function x + the path cost of the node.

Iterative Deepening(ID): (explained in of the void Search method, and in depthLimitSearch method).

-Node depthLimitSearch(SearchProblem p, **int** depth):like the genericSearchAlgorithm it initializes the arraylist of nodes. It then searches for a solution to the problem with a depth limit of (depth). If a node’s depth is equal to depth, it is ignored.

• A discussion of the heuristic functions you employed and, in the case of A\*, an argument for their admissibility.

The first heuristic function is the number of pressure pads left with no rocks on them. The second heuristic function is the Manhattan distance between R2D2 and the teleportal. What makes the two heuristic functions admissible is that in the first heuristic function to reach its goal it will have to go to each pressure pad to push a rock on it and when it can reach its goal the cost becomes zero. In the second heuristic function, for the robot to reach its goal it will have to at least go to the teleportal straight forward. Thus, it will never be exponential or give a cost more than the actual cost.

• A comparison of the performance of the different algorithms implemented in terms of completeness, optimality, and the number of expanded nodes. You should comment on the differences in the number of expanded nodes between the implemented search algorithms.

Important note: algorithms are compared to previously mentioned algorithms only.

Regarding DF, it keeps exploring one path of the search-tree until that path reaches a solution or a dead end, then backtracks and explores the next path. The issue with this approach is that if a path is infinitely long it may explore this path before the path of the solution, which results in expanding many nodes, and wasting time. So DF is not complete as it may run forever before reaching a solution. It may not be optimal in terms of finding the lowest cost solution, or in expanding as few nodes as possible. Worst case expanded nodes are the summation of (b^i) for i=0→(tree depth).

Regarding BF, it explores the search-tree one level at a time, meaning if a solution exists, it will never explore a level deeper than the solution, this is better than DF in the case that a path may be infinitely long, but a solution has to exist otherwise they are basically identical (will explore the entire tree) with the slight difference that DF would use less space (O(depth) vs O(no. Leaves)) i.e. (O(n) vs O (b^n) where b is the expansion factor.) (In short, BF is complete, optimal in that it finds the solution with the least depth first (but not necessarily the lowest cost, our R2D2 problem just has the cost=depth), but not so optimal space wise, or in terms of expanded nodes. if no solution exists it is the summation of (b^i) for i=0→(tree depth). If one does, the expanded nodes are the summation of (b^i) for i=0→(solution depth)

Regarding UC, it explores the search-tree following the shortest path first. If the path of the solution costs an infinitely large amount, UC may end up exploring the entire search tree which may be infinitely long so it is not complete (but in our search problem, R2D2, path cost is the same as depth so UC is equivalent to BF). It is optimal in that it finds the solution with the lowest cost. In terms of space, it is identical to BF as each edge has cost of 1, and in general the queue of nodes should not exceed b^n size. The expanded nodes however may be the entire search-tree. I.e are the summation of (b^i) for i=0→(tree depth).

Regarding ID, it uses depth limit search which explores the search-tree similar to DF, but ignores paths deeper than a certain depth. ID keeps incrementing the cut-off depth and attempting the DL search again. This helps with DF’s disadvantage when a solution exists and an infinite path is explored before it. But since no paths are infinite DF won’t run forever. On the other hand, ID keeps incrementing the cut-off depth and searching again, which will in fact run forever in case no solution exist. This could be altered slightly by using a flag to indicate when the deepest path in the DL search wasn’t deeper than the actual Limit, but no solution was returned (this idea was not implemented in our project in time unfortunately). So, regarding completeness, it is complete if there is a solution the worst case is exploring the entire tree but does not terminate if no solution exists. Regarding optimality, like BF it finds the solution with the lowest depth first, but not necessarily the lowest cost. Regarding the number of expanded nodes, it is extremely inefficient as expanded nodes are summation of j=0→ infinity (summation of (b^i) for i=0→(j)).

Regarding GR1, (Manhattan distance) it explores nodes closer to the teleportal first. Which may not be optimal in terms of expanded nodes or solution cost as sometimes moving closer the teleportal pushes a rock into a corner. Also it may not be complete if infinitely long paths exist. Expanded nodes are the same worst case as DF.

Regarding GR2, (pressure pads left) it explores nodes with less unactivated pressure pads first. Which may not be optimal in terms of expanded nodes or solution cost as sometimes pushing a rock onto a pressure pad may block another rock from moving. Also it may not be complete if infinitely long paths exist. Expanded nodes are the same worst case as DF.

Regarding A\*x, same as GRx it may not be the most optimal solution but it may be more optimal than GRx alone in general. But in our case the path cost is equal to the depth as it increases by 1 in each node with respect to its parent so it will be similar to GRx but with taking the depth in consideration.

• A list of complete and precise instructions on how to run the program and interpret the output.

To run the code, just run the main method from the GenericSearch class and call the Search method with the desired parameters. The output will take one of two forms, if visualize equals to false, the output will be the sequence of operations done (L,R,U,D) starting from the state where the goal test was satisfied going to the initial state. If the visualize is equals to true, the same output as the above plus the grid of each state.