Introduction to Artificial Intelligence

Project 1: Help R2-D2 Escape!

Ahmed Ashraf Awwad
Mohamed Khaled
Omar Radwan

The problem

The R2D2 robot has to find its escape path from the prison through the activated teleportal. The prison contains rocks which can be pushed, unmovable obstacles and pressure pads. In order to activate the teleportal, each rock has to be placed over a pressure pad. The R2D2 robot can move in four directions (North, South, East, West) unless, it hits a rock, obstacle or unactivated teleportal. The R2D2 robot can push the rock in one of the four direction if the next cell is free from obstacles.

Search-Tree Node ADT

The node is a data structure that consists of the state, cost, heuristic cost, depth and parent. The node holds the current state of the problem. The cost represents the cost of the path from the root to the current node in the search tree. Also, the heuristic cost represents the estimated cost from the current state to the goal state. Moreover, the depth represents the depth of the current state from the root.

Search Problem ADT

The problem consists five tuple which are the root node, the state space, the collection of operators, a goal test function and a path cost function. The root node holds initial state of the problem. The state space represents the graph of the set of states reachable from the initial state by any sequence of actions. The collection of operators represents a set of actions that can be done by the agent. A goal test function checks if the given state is the goal state or not. A path cost function retrieves the cost of the given node.

R2D2 Problem

***R2D2Problem* class**: Extends the main *SearchProblem* class. It takes the grid and all initial positions of the robot and rocks as input. It has the following functions and attributes:

- *Final int moveCost* : the *moveCost* which is equal to one (cost of robot step towards an empty cell) .
- *Final int push cost:* which is equal to Five (cost of robot step with pushing a rock).
- *Boolean goalTest():* It takes a node as input and check that the position of the robot in this node is the same as teleportal position. It also checks that each rock is placed on a pressure pad.
- *Boolean valid():* Checks that the robot and the rocks positions are inside the grid and not on an obstacle or an unactivated teleportal.
- *ArrayList<Operator> creatOperators()*: Creates the available actions for the robot. The actions are either moving or pushing the rock in one of the four directions and returns list of available operations.
- *Grid castGrid()*: returns a grid that represents the state space of R2D2 problem.
- *Int heuristicFunction1()*: Compute the heuristic cost according to the distance between the robot and the teleportal.
- *Int heuristicFunction2()*: Compute the heuristic cost based on the summation of the distance between each rock and its closest pressure pad multiplied by the push cost. In addition to, the city block distance between the robot and the teleportal.
- *Int getDistance(Point p1, Point p2)*: Compute the city block distance between two points.

Grid class: implements the main graph/ state space of the search problem. It takes the the length M and width N as parameters. It creates M*N char array according to the following mapping:

Cell description	**Character**
Robot	^
Teleportal	G
Pressure Pad	X
Obstacle	#

Rock	*

- *Void fillRandom()*: It take pads number, obstacles number and teleportal number as an input. Then, it Iterates over grid cells to choose for each cell whether it is free, pressure pad, obstacle or teleportal. After that, it Iterates over the free cells and the pressure pad cells to choose for each cell whether a rock, robot or nothing will be positioned.
- *String charArrayToString()*: Returns a graphical representation of R2D2 space search.
- *String printPath()*: returns a sequence of actions taken in order to reach the given node from the given root of the R2D2 problem.

R2D2State class: Extends the main *state* class. The R2D2 state consists of:

- *Point currCell*: R2D2 robot current cell
- *Arraylist<Point> rocksCells*: the current positions of the rocks.
- *compareTo(State state)*: Compare two states based on the current cell and rock positions.

R2D2Operator class: Implements the main *Operator* class it has the following functions and attributes.

- *Int dx*: The change in the position of row, control the moving up and down in the grid.
- *Int dy*: The change in the position of columns, control the moving left and right in the grid.
- *Int moveCost*: The cost of the operator if it leads to move actions.
- *Int pushCost*: The cost of the operator if it leads to push actions.
- *Node operate(Node node)*: The function tries to move the robot in the operator's direction. If the updated position is occupied by a rock, then the rock position will be updated in the same direction and assign the cost of the operator as the push cost to the new node, Otherwise assign the move cost to new node.
- *Point updatePoint(Point point)*: return the updated point.

Search Algorithms

BFS, DFS, A and Greedy* are similar in the process of finding the goal node, only different in ordering the expanding node in the queue. Initially only the initial node of the problem in the queue. Then, the first node in the queue is removed and if it doesn't pass the goal test and it is not visited before, the algorithm try to expand it using the problem operators, otherwise the goal node will be returned. The algorithm keeps running until it finds a goal node or the queue has no nodes to expand.

In case of *Iterative depth*, The function iterate with a counter from 1 to MAX by step 1. The counter represents the threshold depth for discovering nodes. Each iteration discovered the nodes as discussed in DFS. In case that the selected node from the queue has depth more than the threshold, It will be removed. If *DFS*-like process never found a node that have depth more than then current threshold, then the problem hasn’t solution.

Each created node has two different costs: *actual cost and heuristic cost*. *Actual cost* represent the path cost from the root to this node. *Heuristic cost* is assigned according to a heuristic function that take the state and estimate the cost of reaching the goal from it.

Querying function behavior according to the type of the algorithm strategy:

- *Breadth first search (BFS)*: In order to preserve expanding the nodes level by level in the breadth first search, the queuing function orders the nodes in the priority queue based on the minimum depth.
- *Depth first search (DFS)*: It expands the deeper node first by ordering the nodes in the priority queue based on the one with maximum depth comes first.
- *Iterative depth (ID)*: It is the same as DFS queuing function.

- *Uniform Cost (UC)*: The node with the minimum actual cost, will be in the front of the queue.
- *Greedy (GR)*: The queue keeps the node with the minimum heuristic cost first.
- A* (AS): In the queue the node with the least summation of heuristic and actual cost comes first.

Heuristic Functions

Both *A* and greedy* search strategies depend on the same heuristic functions that will be discussed later. We will discuss two heuristic functions that are related to the *R2D2Problem*. The distance between the cells of the grid is the city block distance.

- *int heuristicFunction2(Node node):* For each rock we compute the distance to the nearest available rock, and make the selected pressure pads not available. Let the sum of all the distances be *totalDist. Formally,* H2(Node) = *totalDist * PushDistance* + *distance(Robot, Teleportal).* We will discuss its admissibility in the only three possible cases:
    - *Goal state: N = 0 and* distance between Robot and Telepotal is zero, thus*, H2(Node(GoalStat)) = 0.*
    - *Semi-Goal state:* when the teleportal is activated and the robot need to go to the teleport cell. *N* = 0 and distance between Robot and Telepotal is output. To prove it is not admissible, we need to prove that there is a shortest distance than city distance are allowed to the robot (diagonal move, jump, …), which is not the case.
    - *Any other state:* We at least need to move the rocks to the pressure pad with city block distance which is guaranteed to be done with similar cost of the function or higher.

- *int heuristicFunction1(Node node):* For each rock we compute the distance between it and the nearest pressure bad. Let *minDistance* is the minimum positive distance between all the computed. Let *N* be the number of rocks which are not positioned on a pressure pad. The estimate cost for the node is basically moving each wrong placed rock to a pressure pad and move to the teleportal. Formally, H1(Node) = *minDistance * N * PushDistance + distance(Robot, Teleportal).We can build our prove on the same arguments of the first function. If H2 is admissible then H1 is admissible too.*

A comparison of the performance of the different algorithms

Search Algorithm	**Total Cost**	**No.Expanded Node**	**Complete**	**Optimal**
DFS	125	648	True	False
BFS	17	98	True	True
ID	19	783	True	False
UC	17	111	True	True
GR1	67	807	True	False
GR2	17	48	True	False
AS1	17	101	True	True
AS2	17	49	True	True

Complete and precise instructions on how to run the program and interpret the output.

1. The user choose one of the search strategies to solve the problem and write the representative string for the selected search strategy in the console:
    - “BF” for breadth-first search.
    - “DF” for depth-first search.
    - “ID” for iterative deepening search.
    - “UC” for uniform cost search.
    - “GR” + *i* for greedy search, with $i \in \{1, 2\}$ distinguishing the two heuristics.
    - “AS” + *i* for A $*$ search, with $i \in \{1, 2\}$ distinguishing the two heuristics.
2. The user specify if he need to visualize the solution if exist by replying with “yes”.
3. If the solution exist the output will be the cost, the depth of the reached goal node and the number of expanded nodes through the process of finding solution. In case of visualizing the output, a sequences of grid describe the path of reaching the goal from the initial state.