Introduction to Artificial Intelligence
Project 1: Help R2-D2 Escape!
Report

Ahmed Soliman
Ahmad Elsagheer

October 19, 2017

Contents

1 Introduction

A grid of size $N \times M$ that consists of N rows and M columns is given. The grid contains a robot, a teleport, obstacles, rocks, pressure pads and empty cells. In one move, the robot can do one of the following actions:

- move to an adjacent empty cell. Two cells are adjacent if they share a side.
- push a rock from one cell to an adjacent cell. The robot must be in a cell adjacent to the cell containing the rock and the new cell of the rock must be either an empty cell or a free pressure pad.

The two actions have different costs. The robot must put every rock on a pressure pad, then go to the teleport cell which is not activited unless the first condition is satisfied. The task is to find a sequence of actions that minimizes the total cost which the sum of the costs of the sequence actions.

The aim of the project is to investigate and compare six different search strategies which are: *Breadth-First Search, Depth-First Search, Iterative-Deepening Search, Uniform-Cost Search, Greedy Search* and *A* Search*. The last three strategies are investigated with three different heuristic functions.

2 Implementation Overview

We used java in our implementation. Our implementation is mainly divided into two parts:

- Generic functions and structures that solves any search problem if the necessary information is provided.
- Specific functions and structures related to R2-D2 problem.

The structure of the project is show in the figure below. The following sections will describe each component of the project.

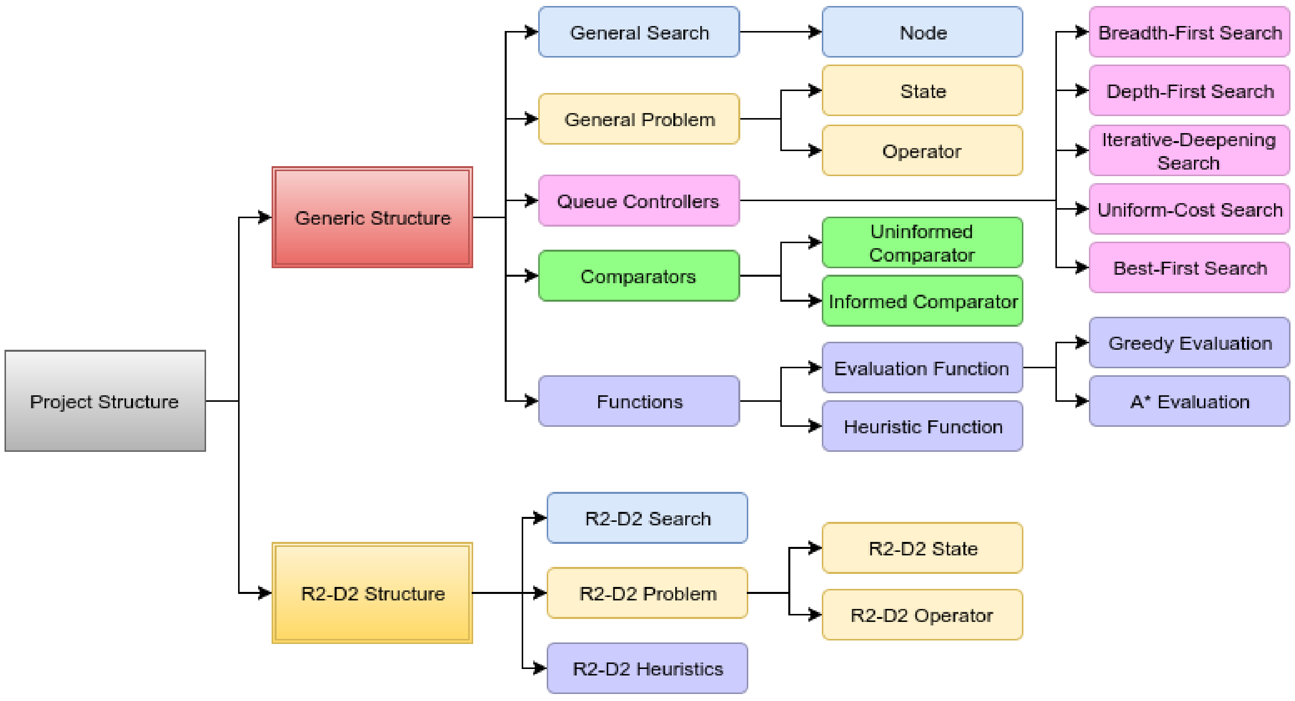


Figure 1: Project Structure

3 Generic Structure

3.1 Problem ADT

Generally, a problem is defined by a set of states called *search space*, an initial state, some goal states and a transition function that maps every state to the states reachable from this state by applying an operator to that state. Our implementation guarantees that any problem that inherits `Problem` ADT must define three things:

- `getInitialState()`: the initial state of the problem.
- `getOperators()`: the set of operators that can be applied to each state.
- `testGoal(State state)`: a test goal function that checks whether a state is a goal state or not.

3.1.1 State ADT

States represent the result of applying a sequence of operators. `State` ADT defines:

- `compareTo(State)`: a state can be compared to another state so that we can detect equal states.
- `toString()`: a representation function so that we can extract information from it.

3.1.2 Operator ADT

Operators are the actions that can be applied to the nodes expand them and get new nodes. `Operator` ADT has:

- `int cost`: the cost of applying this operator.
- `String name`: the operator name to visualize the sequence chosen by the search algorithm to reach the goal state.
- `apply(Node node)`: a function that is applied to a node an returns a new node which is the result of applying the current operator.

3.2 General Search

3.2.1 Search-Tree Node ADT

The search-tree node class named `Node` has five attributes:

- `State state`: that corresponds to the current node.
- `Node parent`: the parent node from which the current node was reached.
- `Operator operator`: the operator which was used to get to the current node.
- `int depth`: the number of operators applied from the initial node to reach the current node.
- `int pathCost`: the cost of the path followed from the initial node to the current node.

3.2.2 Generic Search Algorithm

The generic search is the core part of the project which uses all other instances and functions to solve a given problem. It is implemented in `GeneralSearch` class and has the following methods:

- `expand(Node node, ArrayList<Operator> operators)`: expands the input node using the list of given operators.
- `search(Problem problem, QueueController queueController, boolean visualize)`: searches for a solution to the input problem given a queue controller. It also prints the solution if `visualize` is set to true.

3.3 Queue Controllers

Queue controllers are the queuing functions that determines the order of nodes expansion. The `QueueController` ADT guarantees that all subclasses implement the following methods:

- `makeQueue(Node node)`: constructs a new queue with the initial node.
- `add(ArrayList<Node> nodes)`: adds the input expanded nodes to the queue if they are not visited before. Here, we must note that any search strategy that inherits this ADT is complete, because `QueueController` never visits a state twice and the algorithm will never be stuck in an infinite loop.
- `removeFront()`: removes the front of the queue which is the node chosen for relaxation.
- `isEmpty()`: checks whether the queue is empty.

The implemented queue controllers are listed below. The first three do not take the path cost or any evaluation function into consideration.

- `Breadth-First Search (BFS)`: uses FIFO strategy (i.e. first in first out).
- `Depth-First Search (DFS)`: uses LIFO strategy (i.e. last in first out).
- `Iterative-Deepening Search (IDS)`: uses LIFO strategy. However, what makes it different from `DFS` is that it only allows expansion for nodes whose depth doesn't exceed a certain threshold. When a solution is not found and the queue is empty, the threshold is incremented by 1 and the initial node is added to the queue. This way, if the problem does not have a solution, then the queue will never be empty and the algorithm will never terminate. Therefore, we set a sufficiently large limit for the maximum threshold.
- `Uniform-Cost Search (UCS)`: orders the nodes according to the their path costs. A node with smaller path cost is preferred over a node with higher cost. This comparison is defined through `UninformedNodeComparator` class.
- `Best-First Search`: orders the nodes according to an evaluation function. A node with smaller value for the evaluation function is preferred over a node with higher cost. This comparison is defined through `InformedNodeComparator` class.

3.4 Functions

3.4.1 Heuristic Function

Heuristic functions compute the estimated cheapest cost from a certain state to a goal state. These functions are used in informed search to make the search algorithm pick the node that might be closer to a goal state. `HeuristicFunction` abstract class guarantees that any subclass has a method `apply(Node node)` that returns an integer which is the value of the heuristic function of the input node.

3.4.2 Evaluation Function

Evaluation functions compute estimates for nodes to reach a goal state. Any `EvaluationFunction` must have an attribute `HeuristicFunction heurFunc` which is always part of computing the evalution function. Let n be the current node, $f(n)$ be the evaluation cost of n, $g(n)$ be the path cost of n and $h(n)$ be the heuristic cost of n. We have two types of evaluation functions:

- `GreedyEvaluationFunction`: where $f(n) = h(n)$
- `AStarFunction`: where $f(n) = g(n) + h(n)$

4 R2-D2 Structure

4.1 R2-D2 Problem

`R2D2Problem` class is a subclass of `Problem` ADT. it defines the problem that we are interested in by having the following attributes and methods:

- `int n`: the number of rows in the grid.
- `int m`: the number of columns in the grid.
- `char[][] gird`: 2D array of the grid according to the character mapping shown in table 1

Cell Description	Char	Cell Description	Char
Robot	R	Robot on pad	r
Teleport	T	Robot on teleport	A
Obstacle	#	Empty cell	.
Pressure pad	P	Rock	O
Rock on pad	X		

Table 1: Character Mapping

- `R2D2Problem(int n, int m)`: a constructor that takes the grid dimensions n and m and initializes a random grid and problem operators.
- `R2D2Problem(char[][] grid)`: a constructor that takes the grid to be set for the current problem. This is mainly used to test specific grids. It also initializes the problem operators.
- `initializeGrid(int n, int m)`: initializes the grid as 2D character array with all cells empty. The grid is then populated with rocks, pressure pads, robot and teleport randomly. Preserving the border cells and two cells for robot and teleport, the number of rocks is a random number between 0 and half the number of the remaining cells (the number of rocks is equal to the number of pressure pads). Finally the number of obstacles is equal to random number between 0 and the number of remaining cells. Finaly the cells are populated randomly with symbols according to the assigned character of each symbol. We kept the border cells empty to reduce the probability that we will get a problem with no solution.
- `initializeOperators()`: initializes four `R2D2Operator` instances which are ("Go North", "Go East", "Go South", "Go West").
- `getInitialState()`: returns the initial state of the problem which the initial grid.
- `testGoal(State state)`: tests whether the input state is a goal state. As defined in the introduction, the goal state is the state with the robot on the teleport. It is worth mentioning that `R2D2Operator` takes care that the robot never makes an invalid move. This means that when the robot is on the teleport, then all the rocks have already been moved on pressure pads.
- `toString()`: returns a string containing the grid representation as defined in the character mapping.

4.1.1 R2-D2 State

`R2D2State` is a subclass of `State` ADT having the following attributes:

- `char[][] grid`: 2D grid of the current state.

- robotPosition: the robot position in the grid.
- remainingRocks: the number of remaining rocks that are not put on pressure pads.

The last two attributes can be inferred from the grid. However, they are added to the state to reduce the computing time every time we need to use them.

4.1.2 R2-D2 Operator

`R2D2Operator` class is a subclass of `Operator` ADT having the following attributes and methods:

- `int dx`: the movement in x direction, it is either 1, 0 or -1.
- `int dy`: the movement in y direction, similar to `dx`.
- `int cost`: the cost of this operator which is the base cost of the move. However, if the robot is to push a rock, an additional cost is added to the node path cost. This cost is defined in `R2D2Operator.ROCK_PUSH_COST`.
- `String name`: the name of this operator ("Go North", "Go East", "Go South", "Go West"). The combination of attributes (`dx`, `dy`) defines the direction of movement of the robot as follows: NORTH: (0, -1), SOUTH: (0, 1), EAST: (1, 0) and WEST: (-1, 0).
- `apply(Node node)`: applies the operator on an search-tree `Node`. Simply, it extracts the `R2D2State` from the node and returns a new node with the new state after applying this operator according to the description of the problem and preserving the rules stated. In case of invalid operation a `null` node is returned.

4.2 R2-D2 Search

`R2D2Search` is the function that actually runs the search for the R2-D2 problem. First it allows the user the enter the grid dimensions or the full grid and creates an `R2D2Problem` instance with the given input. Then, it promotes the user to try different search strategies. When a strategy is specified by the user `R2D2Search` creates the corresponding `QueueController` instance. If the search strategy is *Best-First Search*, then instances of `HeuristicFunction` and `EvaluationFunction` are created as specified by the user. These instances are passed to `Best-FirstSearch` instance. Finally, it runs `GeneralSearch.search()` passing to it the problem and the queue controller.

4.3 R2-D2 Heuristics

We used three different *admissible* heuristic functions

4.3.1 Weighted number of remaining rocks

In this heuristic, the heuristic cost of the node is the number of remaining rocks multiplied by the push cost of a rock. The function is admissible because every rock will be pushed at least once. So, the path cost from the current state to the goal state can never be smaller than the heuristic cost.

4.3.2 Weighted cost to furthest rock

Here, the heuristic cost is the Manhattan distance to the furthest rock multiplied by the push cost. The function is admissible because the robot in a way or another will go to this rock and push it at least once.

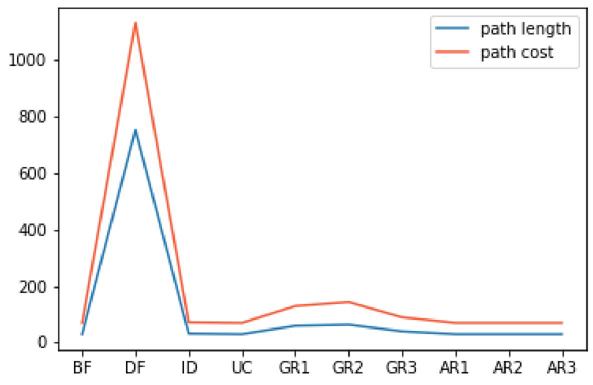
4.3.3 Total Weighted cost of remaining rocks to nearest pressure pads

The heuristic cost of this function is the sum of the Manhattan distances between each rock and the nearest pad to it regardless whether it's empty or not. This heuristic is the most interesting one as it, intuitively, exhibits the largest heuristic cost among the three functions in most of the cases which we expect to expand fewer nodes, in general. This function is admissible with a similar argument. Assuming the best-case scenario that can happen, then each rock will be pushed to the nearest pad to it, no two rocks will share a pad and no obstacle will appear on the way between a rock and a pad. We can see that the push costs summed by the heuristic function is guaranteed to be part of the goal path cost.

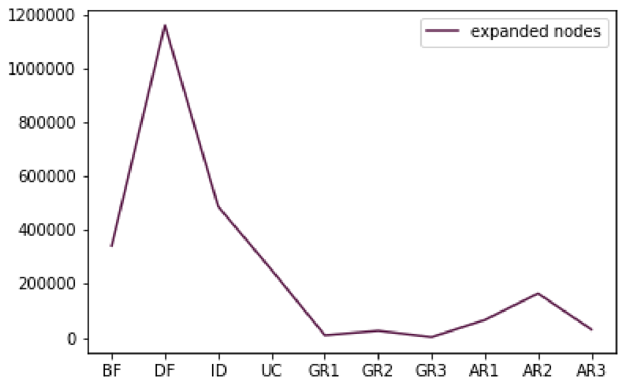
5 Tests and Results

We run all search strategies and plotted their path costs, path lengths and the number of expanded nodes as shown in the two figures below. The values are generated from the case:

```
......
...OP.
.PR.O.
.O#T..
.OPP#.
......
```



(a) Comparision for path cost and path length



(b) Comparision for # expanded nodes

We can see from the figures that:

- BFS guarantees the shortest path length. However, it doesn't always output the optimal solution. If we tried the case below on BFS and UCS, then BFS will output a path length 12 and a path cost 42 while UCS will output a path length 12 and a path cost 22. The number of expanded nodes in the BFS is large.

```
......
..T...
.P..O.
RO..P.
......
```

- DFS doesn't output the optimal path cost and has the largest number of expanded nodes.
- IDS doesn't ouput the optimal path cost as well because it is quite similar to BFS. However, since we do not visit a state more than once, it is also not guaranteed to output the minimum path length.

- `UCS` outputs the optimal solution and has smaller number of expanded nodes than `BFS`. However, there is a log factor in the data structure used in `UCS` which makes it quite slower.
- *Greedy strategy* has extremely the least number of expanded nodes. However, as expected, it is suboptimal.
- *A* strategy* outputs the optimal path cost and has fewer number of expanded nodes than uninformed strategies, but more than *Greedy Search.* As we expected, the third heuristic function has the least number of expanded nodes. It is worth mentioning that it dominates the first heuristic function, but not the second.

  All the search strategies are complete because we augment our `QueueController` ADT with a data structure to check for visited nodes. If we removed it, then only `DFS` will not be complete.

6 How it works

Run the `main()` method in `R2D2Search` class. You will be promoted to enter grid dimensions. If you entered -1 -1, you will asked to enter the full grid as described in the character mapping. Otherwise, a random grid will be created with the input dimensions. Then, you will be allowed to try different search strategies using the following codes:

- `BF`: Breadth-First Search.
- `DF`: Depth-First Search.
- `ID`: Iterative-Deepening Search.
- `UC`: Uniform-Cost Search.
- `GRi`: Greedy Search with 3 different heuristics, $i = \{1, 2, 3\}$.
- `ARi`: A* Search with 3 different heuristics, $i = \{1, 2, 3\}$.

Available heuristics:

- Weighted number of remaining rocks.
- Weighted cost to furthest rock.
- Total weighted cost of remaining rocks to nearest pressure pads.