Help R2D2 Escape!

Ahmed Mostafa Kamel (31-15431, T14), Mostafa Abdullah (31-15451, T14)

The German University in Cairo, Cairo, Egypt

1. Introduction

The goal of the project is to implement a generic search problem solver which, independent of the search strategy applied, searches for a valid path to a goal state. An appropriate queuing strategy is given to this search problem to apply the corresponding search algorithm. In this project, we do a special implementation for the search problem, where the search space is a 2D grid which contains some rocks, rock pads, obstacles, robot and a teleport, and the search goal is to move the robot to put the rocks on the rock pads then go to the teleport avoiding running into any obstacles. The rocks can only be pushed from outside (i.e., the robot cannot stand in the same cell as a rock). This instance of the search problem is tested with various searching algorithms: breadth-first search (BFS), depth-first search (DFS), iterative deepening search, uniform cost search, A* search and greedy search.

2. Search Tree Node

The search tree node is implemented with an abstract class `State`, which contains the following information:

- The search state of the tree node
- The cost to reach this state
- A reference to the parent search node
- The level of the tree node in the search tree

Each instance of the search problem creates its own `State` class which extends the abstract one and holds any information specific to this search instance.

3. Search Problem

Each search problem contains the following abstract components:

- The initial state of the search problem
- A set of valid operators that are applicable to the states of the problem.
  - A cost function which calculates the cost of certain operator is given to each operator.
- The goal tester which tests whether a given state is the search goal or not.
- The search queue which is responsible for the storage and the order of expansion of the search nodes.

4. HelpR2-D2 Problem

We implement a concrete component corresponding to the abstract ones discussed in 3 as follows:

- The initial state is the randomly generated grid, zero cost and zero level.
- There are four valid operators, each corresponding to moving in one of the four main directions: north, east, south and west.
  - The cost of a move is 2 if it results in pushing a rock, otherwise it is 1.
- The goal tester checks if the robot stands in the same cell as the teleport and all the rocks are put on the rock pads.
- The search queue is implemented according to the search strategy used in the problem instance.

5. Main functions

5.1. Main.genGrid

Given the maximum size of the grid `n * m`, it generates the grid with `| 1 : n |` rows and `| 1 : m |` columns. The following objects are randomly distributed across the grid:

- The number of rocks (Let us denote it `r`) and rock pads: a random number in the range `| 0 : (n * m - 2)/ 4 |`
- The number of obstacles: a random number in the range `| 0 : (n * m - 2 * (rocks + 1))/ 4 |`
- The initial position of the robot, the teleport, the rocks, the obstacles and the pressure pads.

5.2. Main.search

- Generates a random grid.
- Creates the search problem with all the components discussed in 3.
- Starts the search algorithm.

5.3. Search.startSearch

The main function in the search algorithm, responsible for expanding the nodes from the search queue according to the used queuing strategy, and detects whether the goal state has been reached.

5.4. Operator.applyOperator

Given a state, it applies this operator on it and gives the next state.

5.5. ExpansionHandler.expand

Given a state, it applies different operators from the search problem on it and enqueus the resulting states in the search queue.

6. Search Algorithms

The search algorithms only differ from one another in the type of the used data structure for storing the search tree nodes:

- Breadth-first Search: A normal queue.
- Depth-first Search: A stack.
- Iterative Deepening Search: A stack like DFS. However, only nodes with level up to the current allowed level are added. When all the allowed nodes are expanded and no solution is found yet, the allowed level is incremented, the queue is cleared and we roll back to the initial state.
- Uniform Cost Search: A priority queue, where higher priority is given the node with the least cost.
- A* Search: A priority queue, where higher priority is given to the node with the least value of the evaluation function `f(n)` given by `g(n) + h(n)`, where `g(n)` is the actual cost taken to reach node `n` from the initial state and `h(n)` is the admissible heuristic cost from the node `n` to the goal state.
- Greedy Search: A priority queue, where higher priority is given to the node with the least heuristic cost (without taking the actual cost into account).

7. Heuristics

7.1. Furthest Rock

The heuristic cost is calculated as the manhattan (city block) distance from the robot position to the position of the furthest rock not on a pressure pad. This function is admissible because in the best condition, the robot needs to go to this rock to push it on a pressure pad. So the actual cost will always be greater than or equal to this distance.

7.2. Remaining Rocks

The heuristic cost is calculated as the number of remaining rocks not on a pressure pad. It is admissible because in the best case, the robot needs to push all these remaining rocks, so the total actual cost will be always greater than or equal to the number of remaining cost.

8. Algorithm Comparison

The following grid is used as the initial state in the comparison between different search algorithms:

```
#.###...
...#....
.*..._#.
._..*M#.
#..|....
```

Where `M` denotes the initial position of the robot, the dot `.` denotes an empty cell, the hash `#` denotes an obstacle, the asterisk `*` denotes a rock, the underscore `_` denotes an empty pressure pad and the bar `|` denotes the teleport.

	BFS	DFS	ID	UC	AS1	AS2	GR1	GR2
Cost	22	146	24	18	18	18	22	22
Exp. Nodes	2748	8918	23888	2613	794	1652	33	188
Completeness	Complete*							

*Theoretically, all the algorithms are complete except DFS, which may keep expanding an infinite branch, whereas the solution may exist in another branch. However, in our implementation, we keep track of any previously visited state and never visit it again. So we make DFS complete.

Incurring a cost `2` for pushing a rock, and `1` for a normal move, the optimal cost for the above grid is `18`. From the above table, it is clear that BFS, DFS, iterative deepening and greedy algorithms are not optimal. As for the number of expanded nodes, BFS, DFS and uniform cost are comparable to each other (they expand close number of nodes). The number of expanded nodes in iterative deepening is much larger because the nodes expanded in a previous iteration are re-expanded in further iterations. One of the A* heuristics is strong that it expands approximately quarter the number of nodes expanded by uniform cost, while maintaining its optimality. The other heuristic is not as strong, but still expands less nodes than uniform cost search. The two greedy heuristics expand much less nodes than any other search algorithm, but this comes at the cost of its optimality.

9. Running

1. Compile and run `PrisonSearch.java` file.
2. Enter the desired search strategy in the console.
3. Enter whether to visualize expanded nodes or not.

If the visualization is enabled, each state is printed one after the other to the standard output in the order they are expanded. A state is represented in the format presented in 8, followed by its cost. If there is a solution, the path from the initial state to the goal state then follows. The path cost and number of expanded nodes are also returned.