BOĞAZIÇI UNIVERSITY

INTRODUCTION TO ARTIFICIAL INTELLIGENCE
CMPE 480

Project 1 Report

Author:
M. Akın Elden
2015402072

4 November 2019

1 Introduction

The project is implemented using Java as the programming language. I created 6 classes to implement the algorithms. The classes and their short descriptions are:

- **Main:** Input taking, file reading operations are performed.
- **Puzzle:** It holds the puzzle map, initial and goal states. Besides it determines the possible moves at a given state.
- **Graph:** It holds the state graph and tracks the search tree. State graph is formed as a 3 dimensional matrix, first dimension for row, second for column and third for orientation. Search algorithms are implemented under this class.
- **State:** It holds the state information of the agent. Each state is determined by agent's location (row,column) and orientation. If the agents orientation is horizontal, then its row value is the left occupied cell. If its orientation is vertical, then its column value is the upper occupied cell. This way each state is uniquely determined using 3 integers: row, column, orientation. Calculations are made regarding this situation. Location and orientation information is assigned when the state is created. It also keeps the predecessor state, previous move, depth of the state at the tree, path cost until that state and visited information. They are assigned when the state is expanded.
- **Node:** It's similar to state but there might be more than one node pointing the same state. Since the state's predecessor etc. is not known until its expanded, nodes are used in queue operations and related information is transferred to corresponding state when the node is explored. All of the information of node class is assigned when it's created.
- **SortedList:** It extends the LinkedList class and overrides the existing *add()* function. It's used as the queue in uniform cost search operations.

I implemented the search algorithms in two different functions: *uninformedSearch()* and *informedSearch()*.

2 Uninformed Search Algorithms

Since the only difference between depth first and breadth first search is how the nodes are added to and pulled from the queue, they both can be performed using the same function with a parameter

determining the queue operation type. Uniform cost search algorithm needs to sort the nodes according to path costs, so the only difference is the need of a sorted queue. I created a SortedList class which extends LinkedList class and overrides the *add()* function. It first finds the correct position and then inserts the node to that position.
addToLast variable determines when the explored nodes are added to queue:

```java
ArrayList<Node> succs = puzzle.getSuccessors(st);
for (int i = 0; i < succs.size(); i++) {
    Node _n = succs.get(i);
    if (stateGraph[_n.r][_n.c][_n.orientation] == null) {
        stateGraph[_n.r][_n.c][_n.orientation] = new State(_n.r, _n.c, _n.orientation);
        if(!addToLast){
            queue.addFirst(_n);
        }
    }
    if (addToLast) {
        queue.add(_n);
    }
}
```

2.1 Depth First Search

I implemented depth first search such that it first adds the ”Left” node to the queue, then ”Up”, ”Right” and ”Down”. Before adding a node to the queue, its corresponding state is checked using state graph. If the state is null, then it’s the first node pointing to that state so the state is created and node is added to queue. If the state is not null, then there is another node pointing to that state in the queue, therefore the node is not added to the queue.
For Depth First Search, *uninformedSearch()* algorithm is called with *addToLast* variable is equal to *false*.

2.2 Breadth First Search

There is no need to check whether there is another node pointing to the same state in breadth first search since state will always be visited by the first explored node and the other node will be skipped since the state is already visited. Therefore, I add the node to queue without checking whether corresponding state is null or not.
For Breadth First Search, *uninformedSearch()* algorithm is called with *addToLast* variable is equal to *true*.

2.3 Uniform Cost Search

I created a *SortedList* class to get the node with minimum path cost from the queue. Path cost to the corresponding state is calculated as sum of predecessor state's path cost and move cost. Since there might be more than one node pointing to the same state and their path cost might differ, I added all explored nodes to queue like in BFS. If the state is already visited, node is skipped. The algorithm finds an optimal solution at the end.
For Uniform Cost Search, *uninformedSearch()* algorithm is called with a *SortedList* as the queue and *addToLast* variable is equal to *true*.

3 Informed Search Algorithms

Like BFS and UCS algorithms, all explored nodes are added to the queue in informed search algorithms. There are two differences between informed and uninformed search algorithm codes:
1. When a successor node is explored, first its heuristic cost is calculated and assigned, then it's added to the queue.
2. Instead of linked list, a priority queue is used as the queue container.

3.1 Heuristics

I used a simple heuristics which returns the Manhattan distance between current state and goal state. It's an admissible heuristics because the Manhattan distance is always less than the real cost for all states. That can be proved like that:
When a state's orientation is single, its next move will go into vertical or horizontal orientation with cost 1. In order to reach the goal state, the agent should turn back to single state with cost 3. All the intermediary moves are taken on the same direction (if the state is horizontal then the intermediary move are vertical otherwise they are horizontal) having cost value equal to 1 and since all the intermediary moves take 1 step, decrease in the heuristic cost is equal to the decrease in the path cost. So the intermediary steps doesn't violate the validity of heuristics, so we can consider them in the proof. Cost of leaving and returning single orientation is 4 excluding the intermediary steps. And the agent can take at most 3 steps. The heuristic cost of 3 step move is 3 but real cost of that action is 4, so the heuristics cost is less than the real cost. Since agent always starts in single orientation and always needs to reach goal state in single orientation, the first move of the agent is leaving the single orientation and last move of the agent is returning to single orientation. As a result, heuristics cost is always less than the real cost path cost and heuristics is a valid and admissible heuristics.

3.2 Greedy Search

Priority queue compares nodes according to their heuristic cost and the node with least heuristic cost is pulled first. If it's visited then skipped to the next node.

3.3 A* Search

The only difference between A* search and Greedy search is that priority queue compares nodes according to the sum of their predecessor states' path cost, move cost (1 or 3) and heuristic cost. Since the heuristic is an admissible heuristic, the algorithm finds an optimal solution at the end..

4 Outputs

Level1

Depth First Search:

```
java -jar proj1.jar levels/level1.txt dfs

42 32 26 26
DRRRRDRDRUULLDRDRUULDRURDL
```

Breadth First Search:

```
java -jar proj1.jar levels/level1.txt bfs

11 55 7 7
RRDRRRD
```

Uniform Cost Search:

```
java -jar proj1.jar levels/level1.txt ucs

10 56 8 8
DRRRRRRD
```

Greedy Search:

```
java -jar proj1.jar levels/level1.txt gs

24 31 16 14
RRDRDRURDLURDL
```

A* Search:

```
java -jar proj1.jar levels/level1.txt as

10 14 8 8
DRRRRRRD
```

Depth first prioritize *Down* move to *Right* move, so the depth first search solution has the highest cost. Also it has the highest depth.
Breadth first search finds the solution with lowest depth since it looks all the states in a depth. However it expanded more nodes than the depth first search. Solution is close to optimal but not optimal.
Uniform cost search found the optimal solution but it has expanded lots of nodes.
Greedy search has found a solution but it's away from being optimal.
Since the heuristics is admissible, A* search found the optimal solution with very few steps.

Level2

Depth First Search:

```
java -jar proj1.jar levels/level2.txt dfs

21 14 13 13
DDLDDLURRDLUR
```

Breadth First Search:

```
java -jar proj1.jar levels/level2.txt bfs

19 33 11 11
DDLLDRRULDR
```

Uniform Cost Search:

```
java -jar proj1.jar levels/level2.txt ucs

19 34 11 11
DDLLDRRULDR
```

Greedy Search:

```
java -jar proj1.jar levels/level2.txt gs

21 16 13 13
DDLDDLURRDLUR
```

A* Search:

```
java -jar proj1.jar levels/level2.txt as

19 29 11 11
DDLLDRRULDR
```

Shape of the map is such that, there is only one path during a few steps, then there the possible states are increases suddenly but the goal state is very close to end of the initial path.
Depth first search found a solution with very few steps. The reason is the shape of the map. It makes search around the goal state.

Breadth first search found a solution with optimal cost. It means that the optimal cost is the first possible solution with the lowest depth.
Uniform cost search found a solution with optimal cost. It's results are very similar to breadth first search.
Greedy search didn't find an optimal solution but it searched few steps.
A* search found an optimal solution and it searched less than uniform cost and breadth first search algorithms.

Level3

Depth First Search:

```
java -jar proj1.jar levels/level3.txt dfs

18 10 10 10
LDLDLURRDD
```

Breadth First Search:

```
java -jar proj1.jar levels/level3.txt bfs

18 32 10 10
LDLDLURRDD
```

Uniform Cost Search:

```
java -jar proj1.jar levels/level3.txt ucs

18 32 11 10
LDLDLURRDD
```

Greedy Search:

```
java -jar proj1.jar levels/level3.txt gs

18 10 10 10
LDLDLURRDD
```

A* Search:

```
java -jar proj1.jar levels/level3.txt as

18 30 10 10
LDLDLURRDD
```

All the algorithms found the optimal solution. The reason might be that, solution is very straightforward. But algorithms expanded different number of nodes until the optimal solution.
Depth first search found an optimal solution without making a backtrack. It means that, goal state can be reached with applying possible moves with the given priority (*Down, Right, Up, Left* in DFS case).

Breadth first search found the optimal solution by expanding all possible states until 10th depth.
Uniform cost search found the optimal solution even by looking at the 11th depth states.
Greedy search found the optimal solution by straightly getting closer to the goal state.
A* search found optimal solution by expanding a little less nodes than BFS and UCS algorithms.

Level4

Depth First Search:

```
java -jar proj1.jar levels/level4.txt dfs

63 91 56 37
RDLURDLURURDLURDLURRRRULLDRUURRRDDDRU
```

Breadth First Search:

```
java -jar proj1.jar levels/level4.txt bfs

33 90 19 19
ULDRURRRRUURRRDDDRU
```

Uniform Cost Search:

```
java -jar proj1.jar levels/level4.txt ucs

33 93 20 19
ULDRURRRRUURRRDDDRU
```

Greedy Search:

```
java -jar proj1.jar levels/level4.txt gs

37 26 21 21
RURRRURDLUURRRDDRRDLU
```

A* Search:

```
java -jar proj1.jar levels/level4.txt as

33 84 19 19
ULDRURRRRUURRRDDDRU
```

There are two bridges in the map and agent needs to use these bridges to reach goal state. So the search becomes first finding the bridges, and then reaching the goal state.
Since there are two bridges, depth first search algorithm may need to make a lot of backtracks, therefore it found a bad solution with a very big depth.
Breadth first search found the optimal solution since it can successfully find the optimum path to bridges.
Uniform cost search found the optimal solution but it has expanded lots of nodes.

Greedy search found a solution close to optimal with much fewer steps than others. The reason is that, bridges can be easily reached by trying to get closer to the goal state in this map. After that it again tries to approach goal state and finds a solution.
A* search algorithm found the optimal solution with less steps then uniform cost and breadth first search. However, the difference is very small.