Assignment 1 - Robot Navigation Problem

Jackson Zaloumis
7671903

Table of Contents

Introduction

Robot Navigation problem is presented on an NxM field, where N & M > 1 (environment). The aim of the robot is to find a route from the starting cell (initial state given as coordinates) to the goal cell (goal state given as coordinates). The robot can only move up, down, left, or right and must only move to blocks that are empty; that is, do not have a wall blocking them and the path cost of movement is 1 unit.

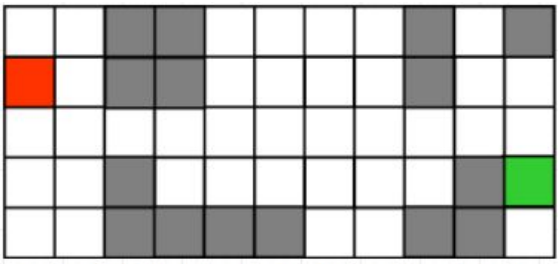
Tree and Graph Searches are implemented to find the solution for the Robot Navigation problem. Search trees are formed "*with the initial state NODE at the root; the branches are actions and the nodes correspond to states in the state space of the problem*"[1]. A frontier stores all the nodes ready to be expanded in the search tree. Search Algorithms get these nodes off the frontier and expand them until the solution is found - or all states are exhausted. The key difference between Graph and Tree Searches are that graph searches store nodes that have been visited, while tree searches do not.

Search Algorithms

There are two different types of Search Strategies used in this project - Informed and Uninformed. All searches implemented have a repeated state check; that is, maintain a list of all the expanded states and do checks before adding a node to the frontier that it has not been visited.

Informed Searches use the information available in the state to determine the best node to expand next. Three types of informed searches were implemented in this project; Greedy Best First, A*, and Iterative Deepening A*.Informed Searches use an evaluation function, f(n), to determine which node to explore next. This evaluation must contain a heuristic value (estimation of cost in getting to the goal state, h(n)) for it to be an informed search.

Below is an example Navigation Problem(test.txt) and will be referenced in describing Search Methods.



Example From "Assignment 1 – Tree Based Search" document

Fig 1

Greedy Best First Search(GBFS) uses only a heuristic value in evaluating which node to explore next and it does not evaluate the cost so this search method is not optimised. Below is a sample of the search tree using GBFS for the problem in Fig 1. As we can see in Fig 2, GBFS selects and stores nodes in the frontier based on manhattan distance - h(n).

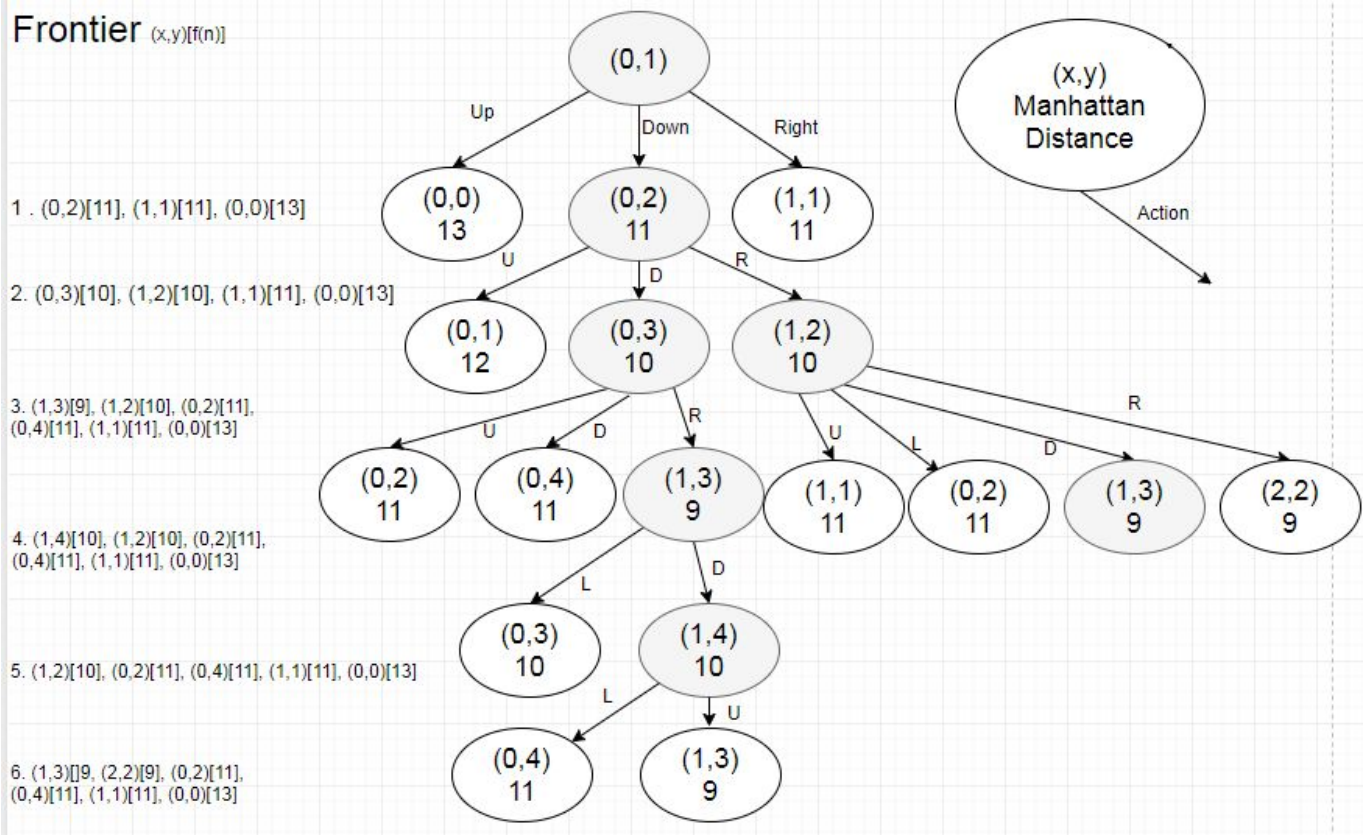


Fig 2 - GBFS Sample Tree

A* Search uses an admissible heuristic (a heuristic that does not overestimate the cost in getting to a node) and the path cost to reach the node, g(n), as its evaluation function in choosing which node to expand next off the frontier.

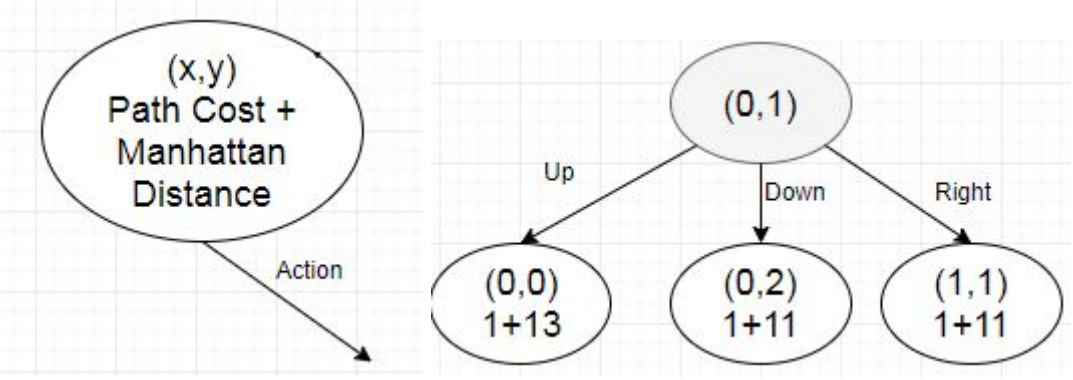


Fig 3 - A* Search Example

In this case, the next node to expand would be (0,2) as it has the lowest evaluation function and it has preference over the right action node. A* continues to expand nodes based off the lowest f(n)=g(n)+h(n). A* continues searching until the lowest possible value of f(n). So even if it has found the solution state, it will continue to search other nodes until the lowest f(n) path cost to solution is found. Below is a small sample of the A* tree diagram for the problem in Fig 1.

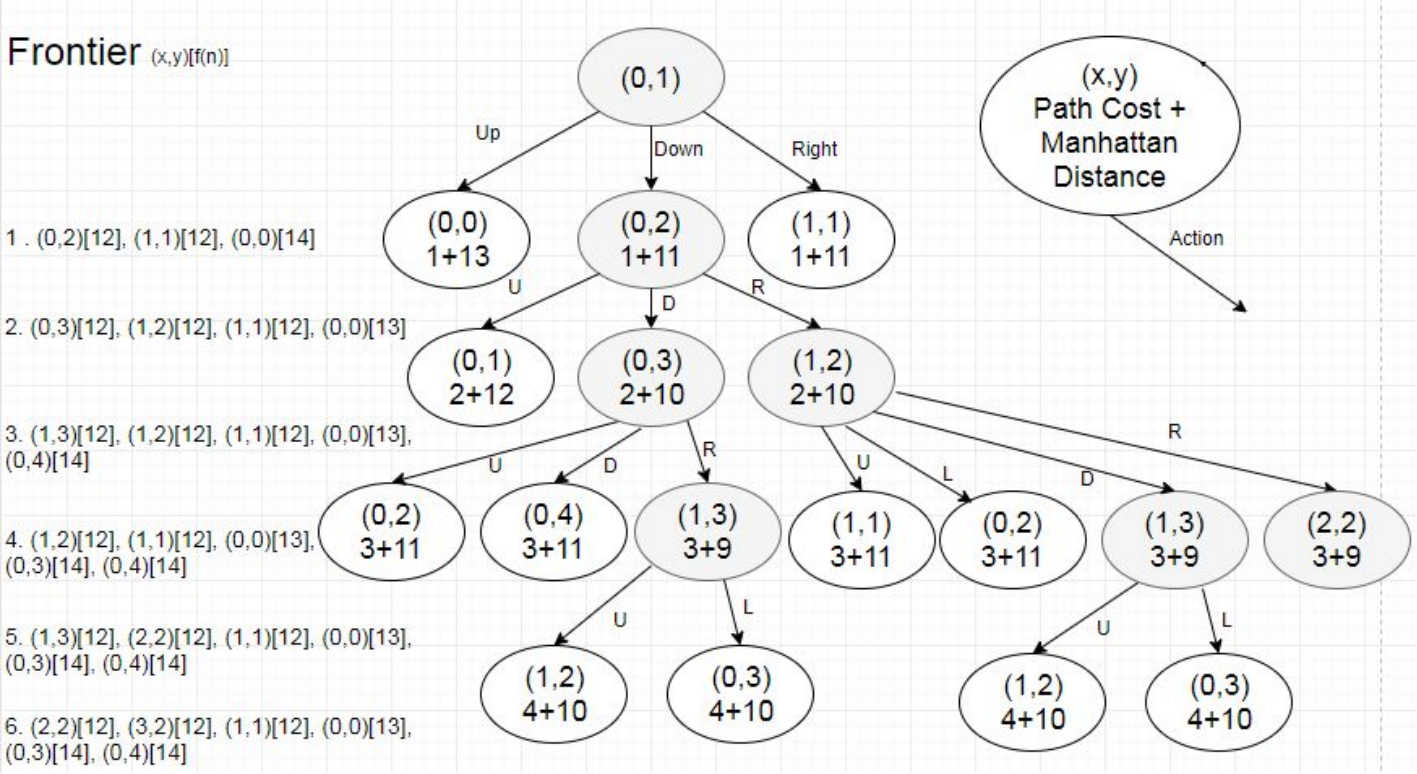


Fig 4 - A* Sample Search Tree

The key difference between A* and GBFS is the evaluation function. A* includes the path cost in f(n), whilst GBFS does not. This difference means that A* can find the optimal solution (with an admissible heuristic) while GBFS is not optimised. As GBFS does not care about current path cost, it will stop searching if it finds the goal state, as f(n) = 0 (lowest possible value). While A*, will continue searching even if it has found the goal state, until f(n) of the goal state is at lowest possible value.

Iterative Deepening A* Search(IDA*) search works similarly to A*; however, if a state is over a threshold value, it is not placed onto the frontier. The threshold value increases if a goal state was not found with the previous threshold value and all states were exhausted. This threshold value is then updated to the minimum f(n) value of all states over the previous value. The state tree is similar to A*; however, it will only search until all states under the threshold value have been exhausted. An advantage of IDA* over A* is that "*avoids the substantial overhead associated with keeping a sorted queue of nodes*"[2]. A downside is the IDA* suffers as it expands the same nodes over and over again as it doesn't store much in memory.

Uninformed Searches are blind searches that do not use any heuristic available in state in finding the solution. Three types of uninformed searches were implemented in this project; Depth First Search, Breadth First Search, and Uniform Cost Search. These search methods are not optimised to find the best solution and can potentially get stuck in loops if implemented without a repeated state check.

Depth First Search(DFS) works by expanding the deepest possible node on the frontier until the solution is found. The frontier works as a Last In, First Out(LIFO) queue and the node to be expanded next is always the deepest node possible. DFS can often get stuck in loops if implemented without a repeated state check and this search method is not optimised.

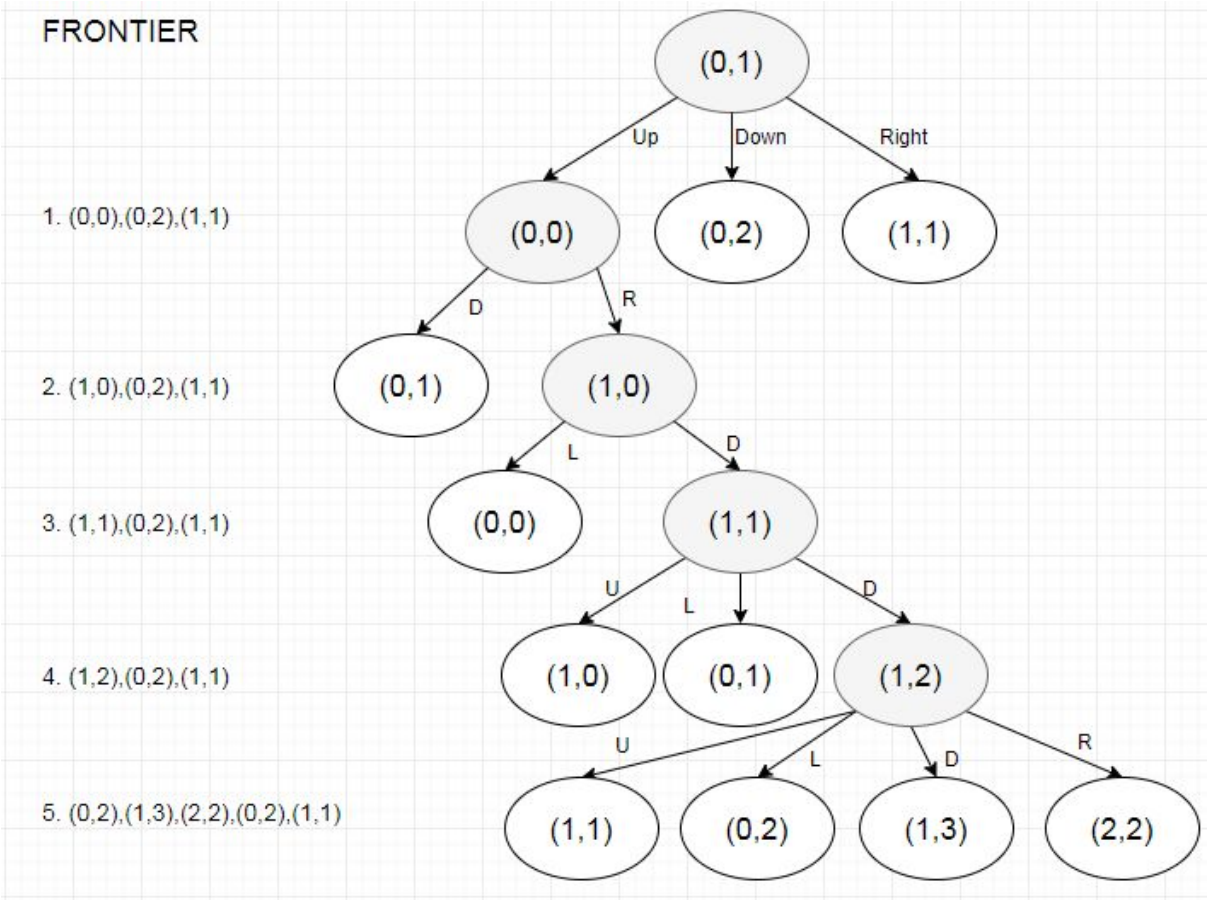
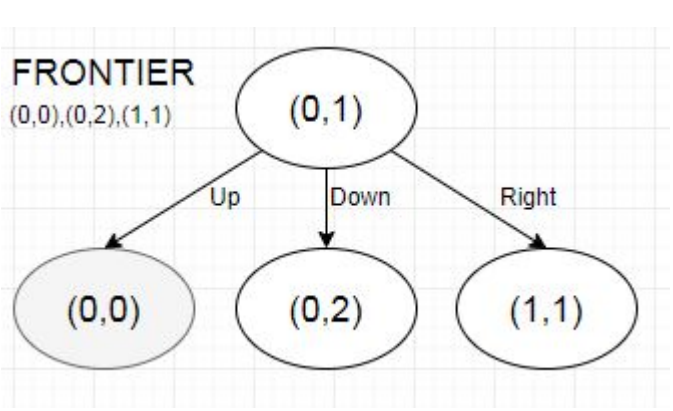


Fig 5 - DFS Search Tree Sample

Above is a sample of the DFS tree for this problem. We can see that the frontier is ordered by the deepest node first, excluding repeated states. DFS will continue its search until a solution is found.

Breadth First Search(BFS) expands the shallowest possible node in the search tree. The frontier for BFS works as a First In, First Out(FIFO) queue with the shallowest nodes at the front of the frontier and the deepest nodes at the end of the frontier and continues searching until a solution is found. BFS can often get stuck in loops if implemented without a repeated state check and although can find the shallowest possible solution this search method is not optimised as it does not care about the path cost.



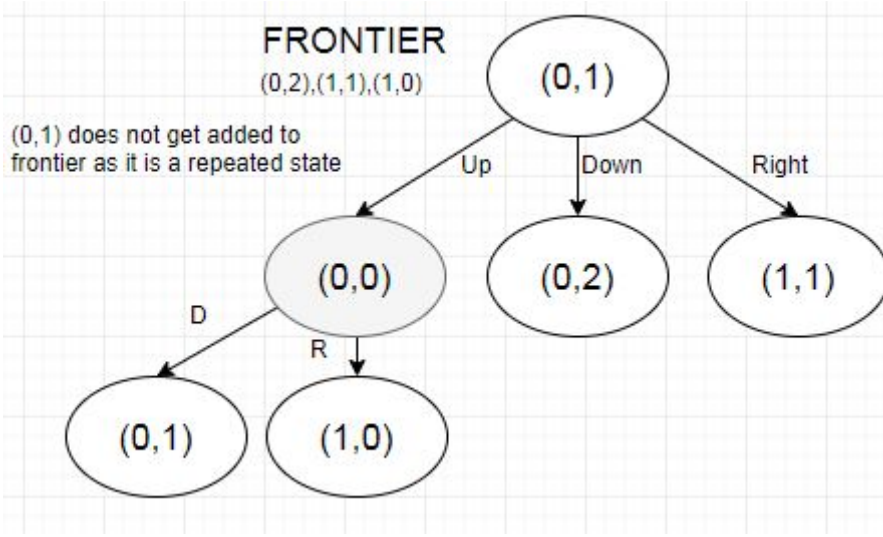
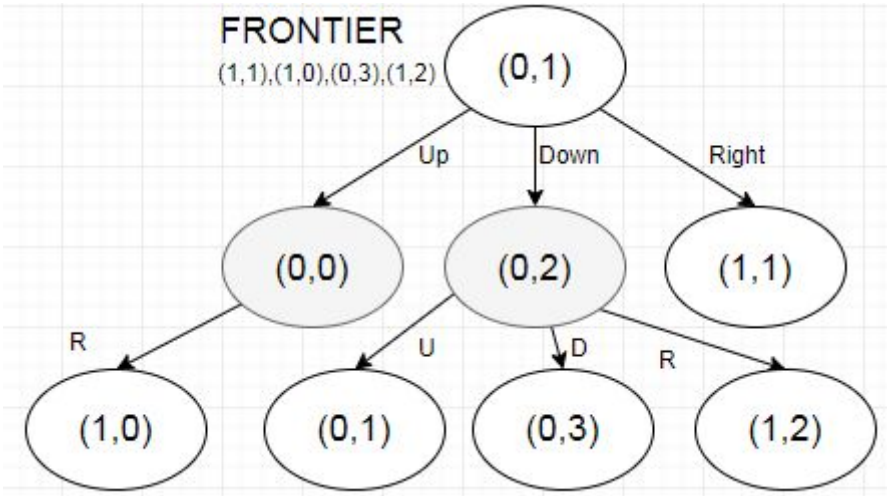


Fig 6 & 7 - BFS Example

Above are the steps taken for a BFS search. We can see that, after it expanded the child nodes are places on the end of the frontier and the next shallowest node is expanded next. Then the next node is expanded on the frontier.



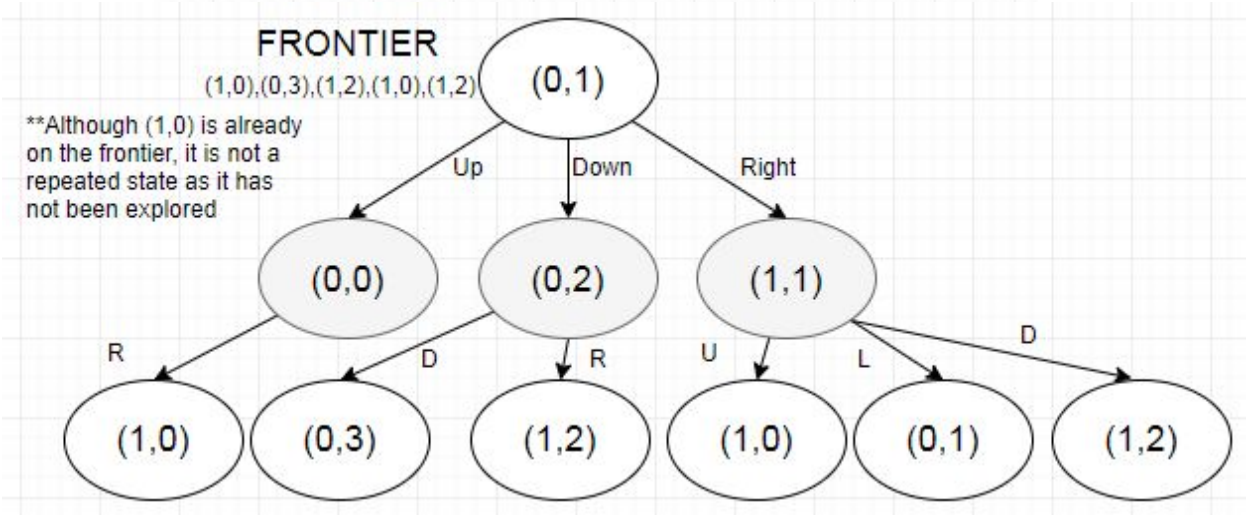


Fig 7 & 8 - BFS Example

After all Nodes have been expanded at the same depth in the search tree, then BFS will move down the tree to the next node on the frontier.

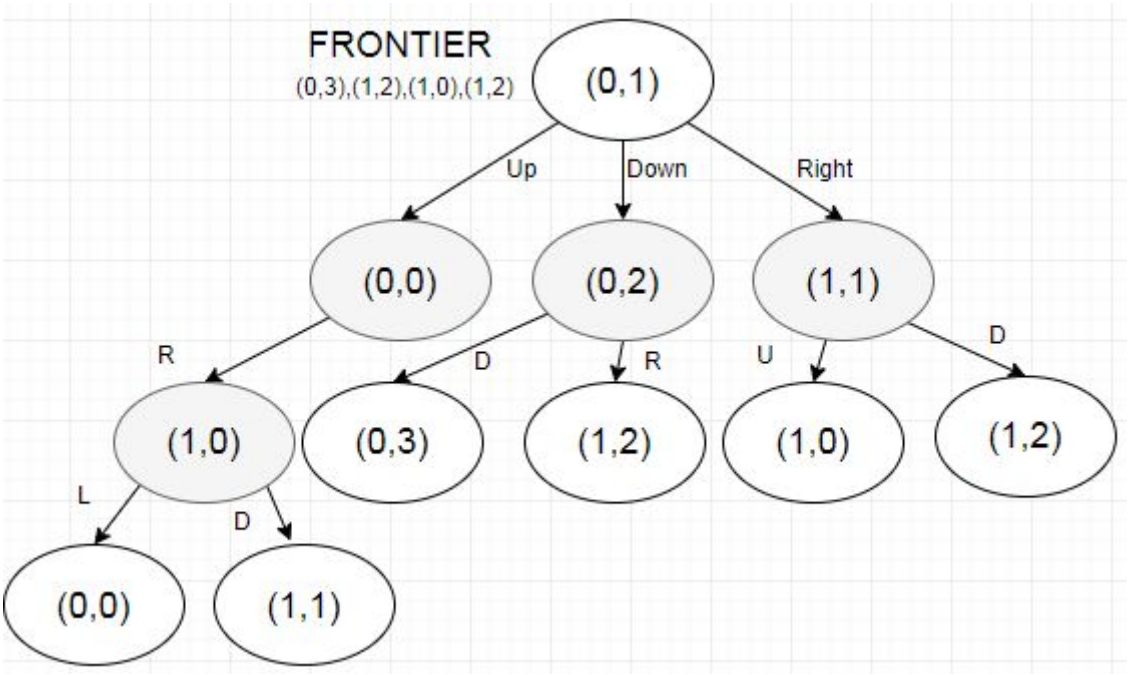


Fig 9 - BFS Search Tree Example

Above is only a sample of the search tree, but BFS continues until a solution is found.

The key difference between BFS and DFS is the ordering of the frontier. DFS orders using a LIFO, which means it expands the deepest possible node first, and BFS uses FIFO, which means it expands the shallowest possible node next.

Iterative Deepening Search(IDS) uses DFS, however limited the depth to which nodes are expanded. If the solution is not found at a certain depth, the depth is incremented at the search starts again until a solution is found or the depth reaches the maximum value(no solution).

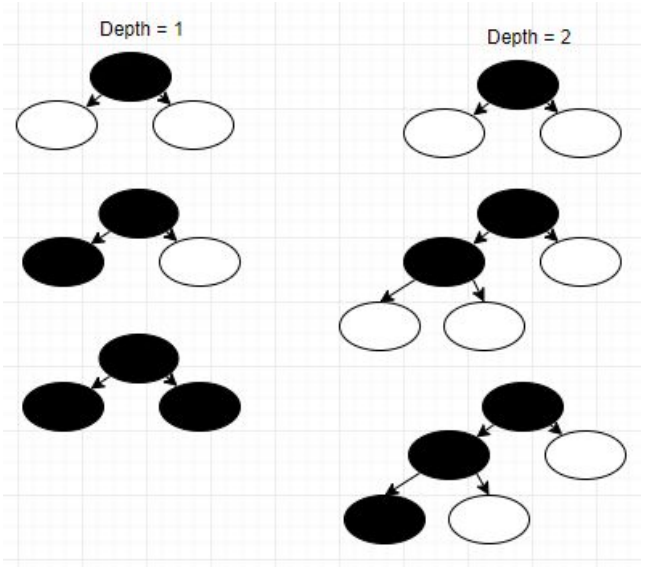


Fig 10 - Example tree of IDS

Test Case Results

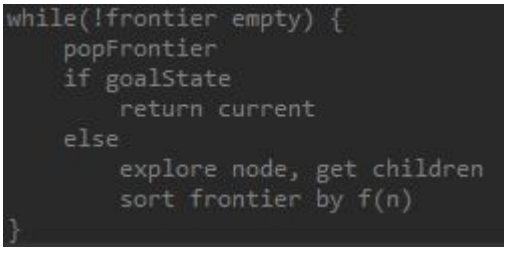
	test.txt		test4.txt	
Algorithm	**Nodes Visited**	**Solution Path Cost**	**Nodes Visited**	**Solution Path Cost**
AS	22	12	53	19
GBFS	18	14	96	31
IDA*	70	12	941	19
BFS	85	12	345	19
DFS	48	30	9538	69
IDS	1432	12	73258	19

From the results of the solution program we can compare efficiencies and the solution path cost each algorithm returns. From the results; A*, IDA*, IDS, and BFS, all return the most optimal solution for both test cases (12 and 19 movements), although BFS and IDS are not an optimised search algorithm (as neither evaluate the path cost). In both examples IDS evaluates the most nodes (1432 and 73258).

Implementation

A* - A* is implemented by having a frontier that is sorted by the evaluation function f(n) = h(n) + g(n) with the node with the lowest value being expanded. A* continues to search until the node on the top of the frontier is at the goal state and has the lowest f(n).

GBFS - GBFS is implemented by having a frontier that is sorted by heuristic with the node with the lowest value being expanded. As soon as GBFS finds the solution (h(n) = 0) it returns the solution. f(n) =h(n) for GBFS.



-Pseudo Code for A* and GBFS

IDA* - Implemented by having an A* search, thats frontier is limited to a certain threshold value. If no solution is found, the threshold value is set to the lowest f(n) of all states over the threshold value.

```
while (no result)
   clear frontier
   result = A*(initialState, cutoff)
   cutoff = min of f(n) values over previous cutoff

return result
```
-Pseudocode for IDA*

DFS - DFS is implemented with a frontier as a LIFO queue which continually expands nodes until the goal state is found.

```
while(!frontier empty)
    popFrontier
    if goal state
        return current
    else
        explore node to get children
        add to Front of Frontier
```
-Pseudo Code for DFS

BFS - BFS is implemented with a frontier as a FIFO queue which continually expands nodes until the goal state is found.

```
while(!frontier empty)
    popFrontier
    if goal state
        return current
    else
        explore node to get children
        add to End of Frontier
```
- Pseudo Code for BFS

IDS - Implemented by having a DFS Search that is limited in depth. When this depth limit is reached with no solution, the depth limit is incremented and they search begins again until a solution is found.

```
for(int depth = 0; depth < MAX_INTEGER; depth++)
  result = DFS(depth)
  if (result != null)
      return result
return null; // no solution found after searching to max integer
```
- IDS Pseduo Code

Repeated State Check Implementation - All search methods use a repeated state, that do not add nodes to the Frontier if they are already on the frontier, or if they have appeared in the current search path (e.g is a parent node with the same Robot Coordinates).

Features/Bugs/Missing

Included Features:

- A* Search,
- Greedy Best First Search,
- Breadth First Search,
- Iterative Deepening A* Search,
- Depth First Search,
- Iterative Deepening Search,
- and Ability to have different costs (denoted by true/false in command line. See readme)

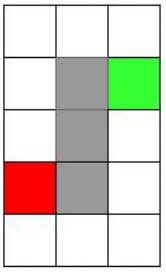
Known Bugs:

- IDA* can not handle a problem with no solution. Will just infinitely loop.
- IDS will loop until Integer.MAX_VALUE in a no solution state

Research

The program has been extended to include different costs for moving a certain direction. A number of Algorithms do not produce a different solution path (BFS, DFS, and GBFS) as none of these algorithms use cost when evaluating which node to expand. The algorithms affected are A* and IDA*. When changing the cost for different movements, the algorithms find a different solution path.

For example, the below set up(test2.txt) with the following path costs (Up = 4, Right=Left=2, Down=1), has differing solutions when compared to all costs being equal to 1.

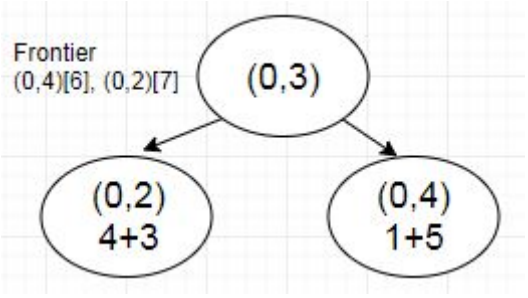


Results of A* search with different costs->

```
C:\Git\AIAssign1\RobotNavigation>RobotNav test2.txt AS true
test2.txt   AS   11
Up;Up;Up;Right;Right;Down;
Total Cost: 6

C:\Git\AIAssign1\RobotNavigation>RobotNav test2.txt AS false
test2.txt   AS   11
Down;Right;Right;Up;Up;Up;
Total Cost: 17
```

Example Diagram with different cost->



From this we can immediately see the impact that a different cost structure has. With a uniform cost, the next node to expand would have been (0,2), but now with a different cost, A* has (0,4) at the top of the frontier even though its h(n) value is larger.

The effect it has on IDA* is that the number of nodes searched is increased

```
C:\Git\AIAssign1\RobotNavigation>RobotNav test2.txt IDA true
test2.txt   IDA   23
Up;Up;Up;Right;Right;Down;
Total Cost: 6

C:\Git\AIAssign1\RobotNavigation>RobotNav test2.txt IDA false
test2.txt   IDA   305
Down;Right;Right;Up;Up;Up;
Total Cost: 17
```

Conclusion

For this problem, the best kind of algorithm to use would be an IDA* or A* search. Both are an optimal search method as long as it has an admissible heuristic so it will always find the lowest cost solution. However, IDA* is not suitable for this problem if the cost is not uniform, as shown in the research. IDA* and A* are more suitable than GBFS, IDS*, DFS, and BFS as all these methods are not optimal so may return a solution with a greater cost than A*.

Acknowledgements/Resources

See readme.txt in submission

References

1. Russel, S. & Norvig, P. 2010, Artificial Intelligence A Modern Approach, 3rd edn, Prentince Hall, p. 75
2. Russel, S. & Norvig, P. 2010, Artificial Intelligence A Modern Approach, 3rd edn, Prentince Hall, p. 99