Fakulteit Ingenieurswese, Bou-omgewing & IT

Faculty of Engineering, Built Environment & IT

EAI 320: Artificial Intelligence

Imtiaz Mukadam

U13083113

Assignment 2: Informed Search Algorithms

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Introduction

After investigating general uninformed search methods we now move on to informed, or heuristic based, searches that are designed to be more specific to its application. The previously investigated brute force algorithms are general, and do not encompass any information about its application. We now incorporate information into the searches to further enhance it in terms of efficiency.

In this task, we are required to implement various informed search methods to guide an AI agent to a goal position in a maze. The searches to be implemented are the:

- Uniform Search – A search that assigns costs for moving from one node to another. The costs of movement are often denoted as $g(n)$. In the given maze, the cost for moving vertically or horizontally is 1 and the cost for moving diagonally is $\sqrt{2}$.

- Greedy Search – this search method uses a heuristic function, often denoted as $h(n)$, that is specific to the application. This function assigns a cost value to each node indicating its closeness to the goal node. The Euclidian Distance will be used as the heuristic for this search problem

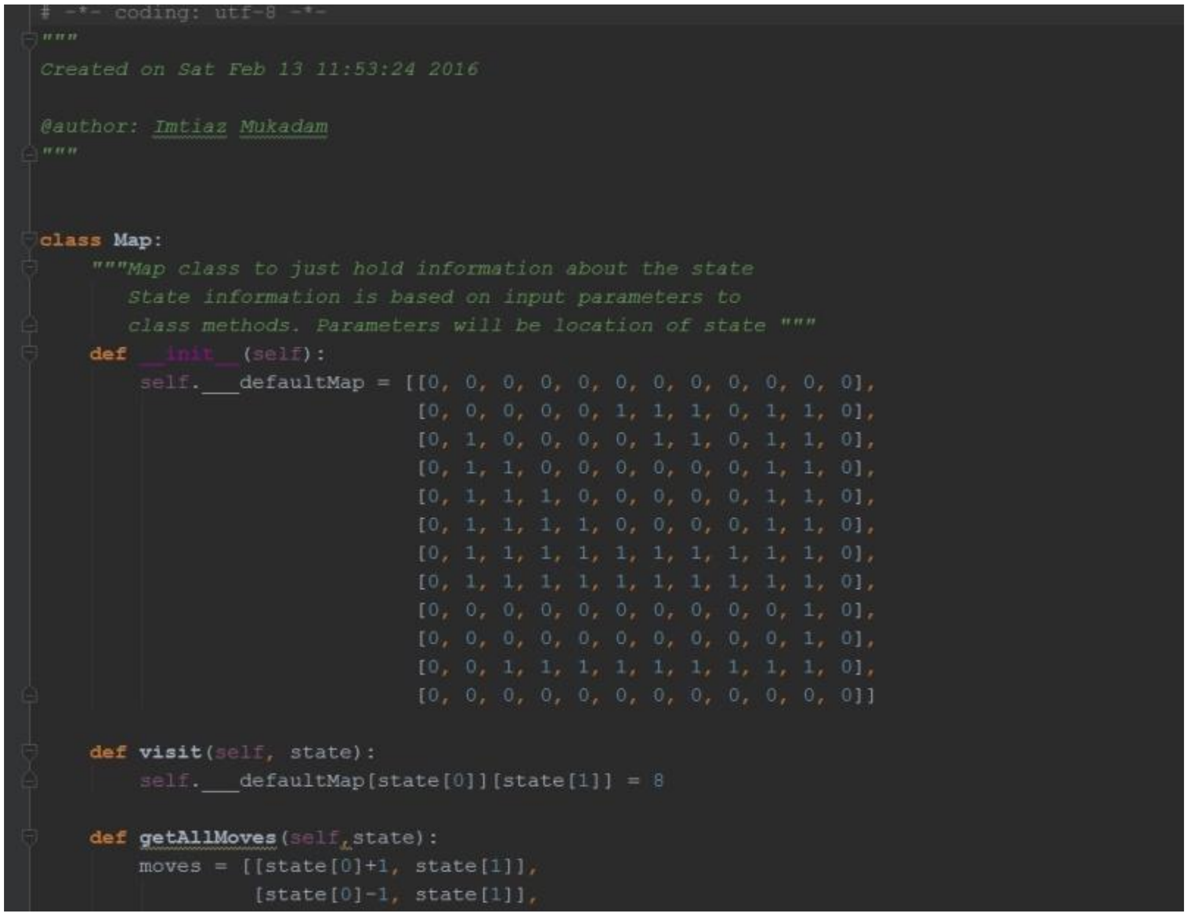
- A* Search – The A* search is a combination of the uniform and greedy search. The A* search uses a cost function denoted by $f(n)$ where $f(n) = g(n) + h(n)$.

These search methods will be investigated under 2 scenarios, in an easy case with no obvious blocks toward the goal node, and in a case where a wall blocks the obvious route to the goal node in the maze.

After the search methods are executed in each case – Their performance and efficiencies will be assessed and compared by keeping track of the number of nodes visited and the path it has found toward the goal node.

Methodology

Using python, a design of the maze will stored and altered as the search algorithm traverses through different maze states. To store the maze, and various functionalities of the maze, a map class is created.



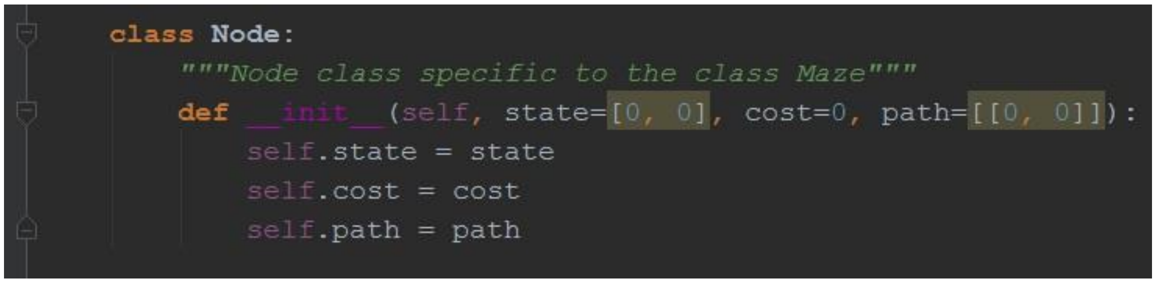
```
# -*- coding: utf-8 -*-
"""
Created on Sat Feb 13 11:53:24 2016

@author: Imtiaz Mukadam
"""


class Map:
    """Map class to just hold information about the state
       State information is based on input parameters to
       class methods. Parameters will be location of state """
    def __init__(self):
        self.___defaultMap = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                              [0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0],
                              [0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0],
                              [0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0],
                              [0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0],
                              [0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0],
                              [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                              [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                              [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
                              [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    def visit(self, state):
        self.___defaultMap[state[0]][state[1]] = 8

    def getAllMoves(self,state):
        moves = [[state[0]+1, state[1]],
                 [state[0]-1, state[1]],
```

Figure 1 Map Class to hold information about the map

The map class contains the default map state and various functions that return valid operations to change states. The map class is created for convenience to organize the code and exclusively handle validating state changes. The map class will also hold the information about which nodes are visited after each search is executed.

For convenience, a number 8 on the map class is used to denote that that state has been added and visited by the search algorithm.

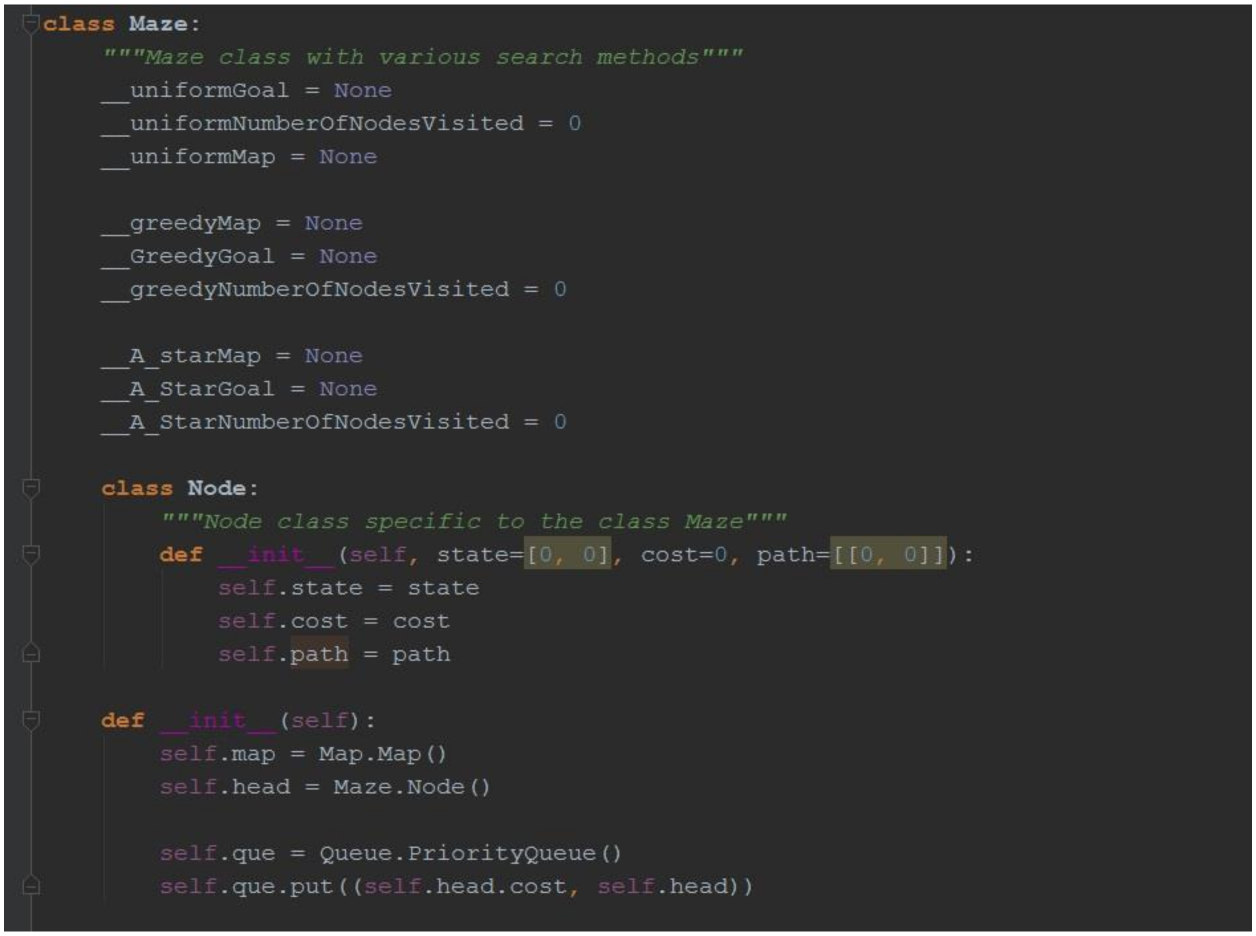
Since any state in the state space only consists of where the player is in a valid position in the map, a state in the node can just be stored as the valid position that the player is currently occupying. Thus the node class can be easily defined in terms of states and costs. The node class also holds information about its path history. In this way, when the goal node is found, both its total cost and information about its path is stored in the node.



```python
    class Node:
        """Node class specific to the class Maze"""
        def __init__(self, state=[0, 0], cost=0, path=[[0, 0]]):
            self.state = state
            self.cost = cost
            self.path = path
```

Figure 2 Node class stored within class Maze

The final class is the maze class itself. The maze class creates an instance of a map and a starting node when instantiated. This class also holds the search methods for each search type.



```python
class Maze:
    """Maze class with various search methods"""
    __uniformGoal = None
    __uniformNumberOfNodesVisited = 0
    __uniformMap = None

    __greedyMap = None
    __GreedyGoal = None
    __greedyNumberOfNodesVisited = 0

    __A_starMap = None
    __A_StarGoal = None
    __A_StarNumberOfNodesVisited = 0

    class Node:
        """Node class specific to the class Maze"""
        def __init__(self, state=[0, 0], cost=0, path=[[0, 0]]):
            self.state = state
            self.cost = cost
            self.path = path

    def __init__(self):
        self.map = Map.Map()
        self.head = Maze.Node()

        self.que = Queue.PriorityQueue()
        self.que.put((self.head.cost, self.head))
```

Figure 3 Maze Class containing search methods

As can be seen above, the class makes use of the PriorityQueue class. A tuple in the form ($cost$, $node$) is entered into the priority queue. The cost passed is the cost stored within the node that will be stored in the queue. In this way, there is no need to sort the queue, and the get() method of the queue will return the node with the smallest cost.

The map class contains the 3 search methods along with various helper functions:

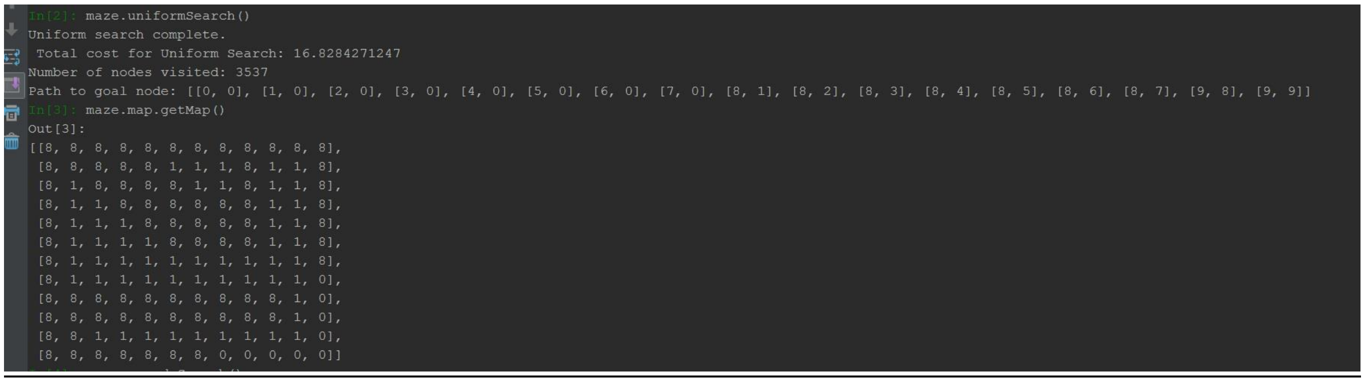
- uniformSearch() – main search method that starts the while loop of the search.
    - peruseUniform() – this helper function visits the node and marks it visited, and ends the search if the goal is found, if not, it calls the
    - expandTreeUniformly() – this function expands the tree by adding nodes to the queue with 1 to the costs of adjacent moves and $\sqrt{2}$ as the costs of diagonal moves.
- greedySearch() – main Greedy search method
    - peruseGreedy()– this helper function visits the node and marks it visited, and ends the search if the goal is found, if not, it calls the
    - expandTreeGreedy() – this function expands the tree by adding nodes to the queue obtaining costs using *map.getHeuristic(state),* which returns the Euclidian Distance of the state from the goal node as the cost of the node.
- A_StarSearch() – Main A*Star search function
    - peruseA_Star() – this helper function visits the node and marks it visited, and ends the search if the goal is found, if not, it calls the
    - expandTreeA_Star() – this function expands the tree by adding nodes to the queue and obtains its cost by using the same *map.getHeuristic(state)* function that was used in the Greedy search. This cost is summed with the step cost of the Uniform search.
- printSearch() – this function prints results of the various search methods after it is executed.

The class also holds various private methods and variables that facilitates for the internal functionality of the code. The code can be viewed within the SOFTWARE folder attached. The code can also be run easily by executing the $search.py$ python file.

Results

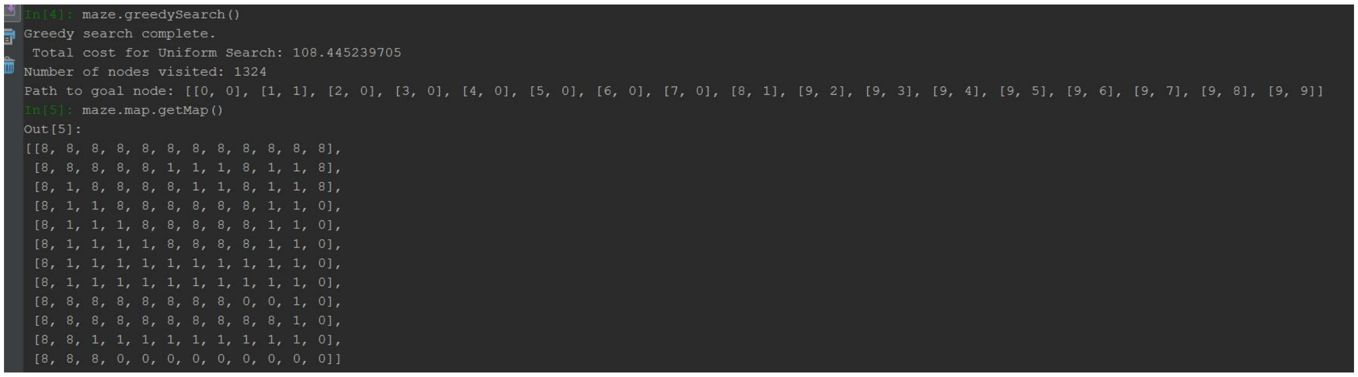
Results are investigated for the case when the (7, 0) is passable and the case where (7, 0) is a wall. Note that a state visited is denoted by the number 8 on the map.

<u>Case 1: (7, 0) is passable:</u>



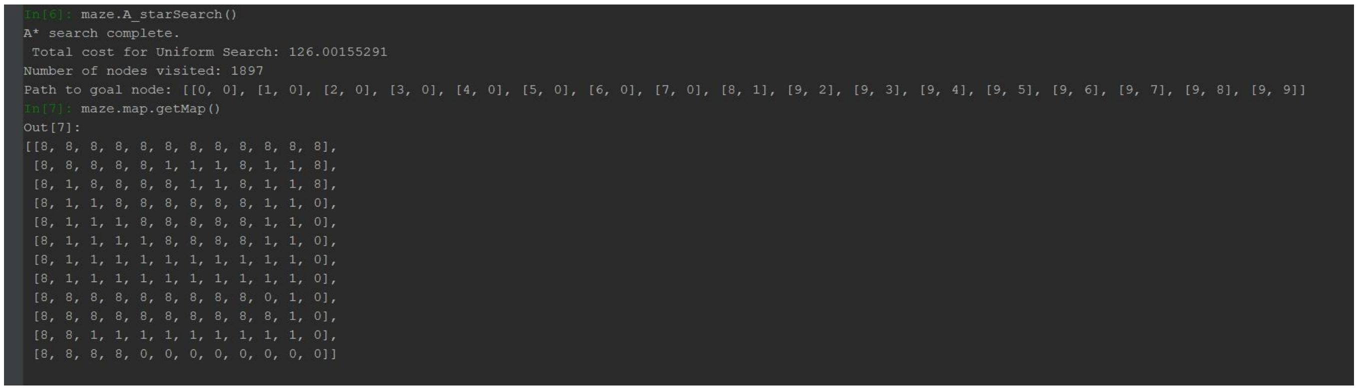
```
In[2]: maze.uniformSearch()
Uniform search complete.
 Total cost for Uniform Search: 16.8284271247
Number of nodes visited: 3537
Path to goal node: [[0, 0], [1, 0], [2, 0], [3, 0], [4, 0], [5, 0], [6, 0], [7, 0], [8, 1], [8, 2], [8, 3], [8, 4], [8, 5], [8, 6], [8, 7], [9, 8], [9, 9]]
In[3]: maze.map.getMap()
Out[3]:
[[8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
 [8, 8, 8, 8, 8, 1, 1, 1, 8, 1, 1, 8],
 [8, 1, 8, 8, 8, 8, 1, 1, 8, 1, 1, 8],
 [8, 1, 1, 8, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 0],
 [8, 8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 0, 0, 0, 0, 0]]
```

Figure 4 Uniform Search with (7, 0) passable



```
In[4]: maze.greedySearch()
Greedy search complete.
 Total cost for Uniform Search: 108.445239705
Number of nodes visited: 1324
Path to goal node: [[0, 0], [1, 1], [2, 0], [3, 0], [4, 0], [5, 0], [6, 0], [7, 0], [8, 1], [9, 2], [9, 3], [9, 4], [9, 5], [9, 6], [9, 7], [9, 8], [9, 9]]
In[5]: maze.map.getMap()
Out[5]:
[[8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
 [8, 8, 8, 8, 8, 1, 1, 1, 8, 1, 1, 8],
 [8, 1, 8, 8, 8, 8, 1, 1, 8, 1, 1, 8],
 [8, 1, 1, 8, 8, 8, 8, 8, 8, 1, 1, 0],
 [8, 1, 1, 1, 8, 8, 8, 8, 8, 1, 1, 0],
 [8, 1, 1, 1, 1, 8, 8, 8, 8, 1, 1, 0],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 8, 0, 0, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 0],
 [8, 8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 8, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

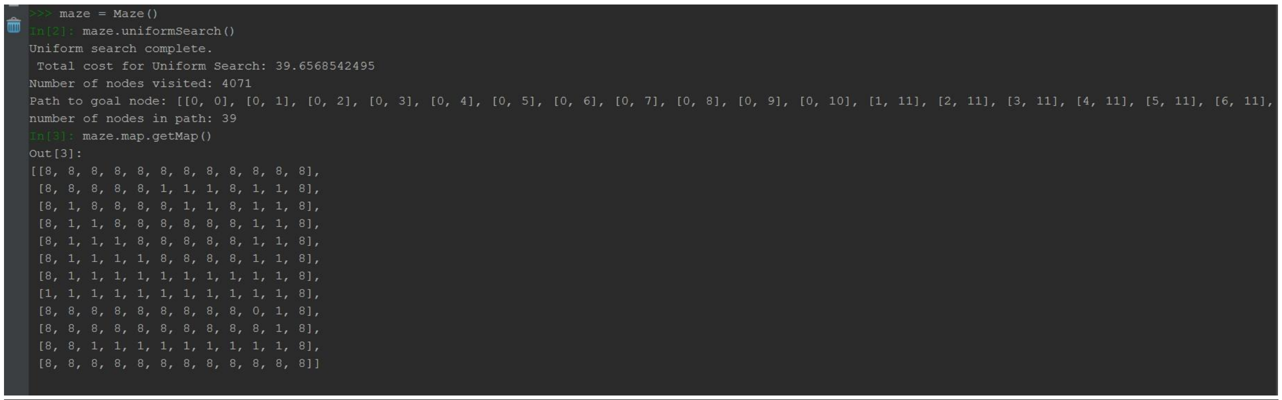
Figure 5 Greedy search with (7, 0) passable



```
In[6]: maze.A_starSearch()
A* search complete.
 Total cost for Uniform Search: 126.00155291
Number of nodes visited: 1897
Path to goal node: [[0, 0], [1, 0], [2, 0], [3, 0], [4, 0], [5, 0], [6, 0], [7, 0], [8, 1], [9, 2], [9, 3], [9, 4], [9, 5], [9, 6], [9, 7], [9, 8], [9, 9]]
In[7]: maze.map.getMap()
Out[7]:
[[8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
 [8, 8, 8, 8, 8, 1, 1, 1, 8, 1, 1, 8],
 [8, 1, 8, 8, 8, 8, 1, 1, 8, 1, 1, 8],
 [8, 1, 1, 8, 8, 8, 8, 8, 8, 1, 1, 0],
 [8, 1, 1, 1, 8, 8, 8, 8, 8, 1, 1, 0],
 [8, 1, 1, 1, 1, 8, 8, 8, 8, 1, 1, 0],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 0, 1, 0],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 0],
 [8, 8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [8, 8, 8, 8, 0, 0, 0, 0, 0, 0, 0, 0]]
```

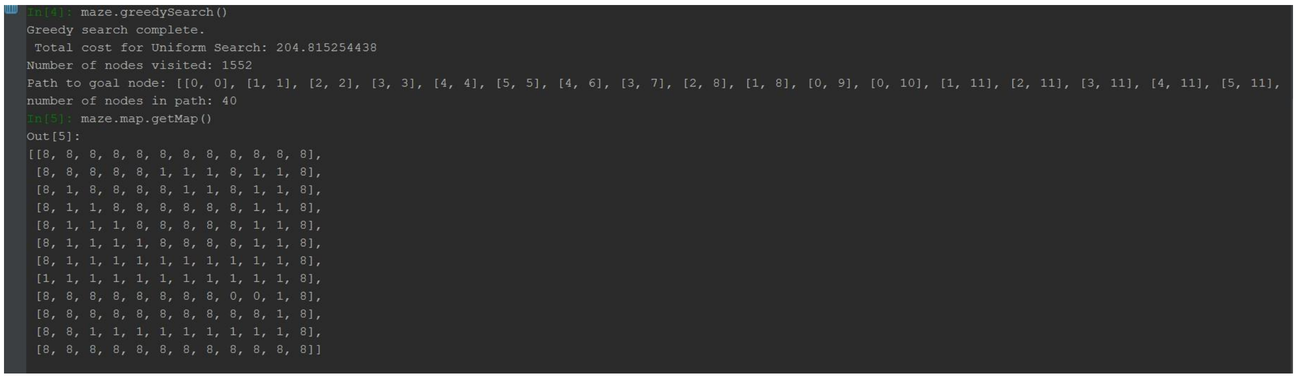
Figure 6 A*Search with (7, 0) passable

Case 2: (7, 0) is a wall:



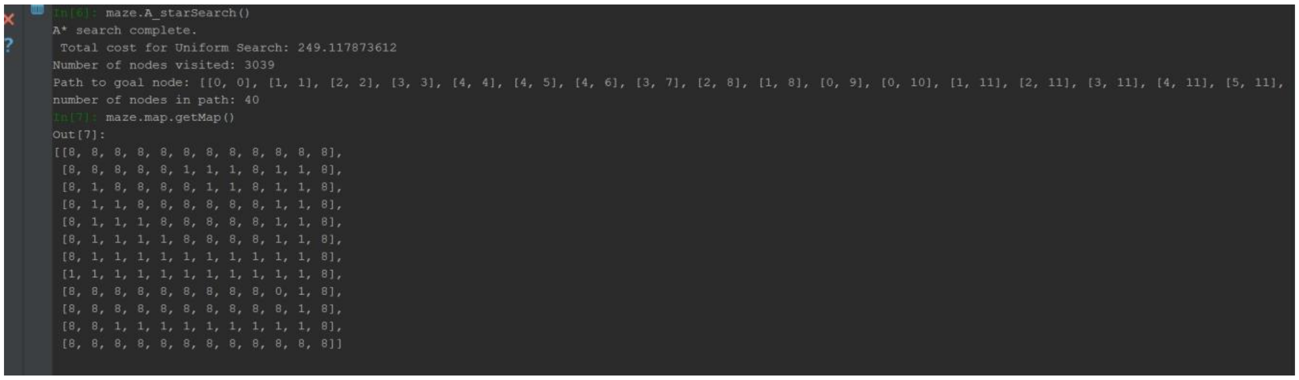
```
>>> maze = Maze()
In[2]: maze.uniformSearch()
Uniform search complete.
 Total cost for Uniform Search: 39.6568542495
Number of nodes visited: 4071
Path to goal node: [[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [0, 8], [0, 9], [0, 10], [1, 11], [2, 11], [3, 11], [4, 11], [5, 11], [6, 11],
number of nodes in path: 39
In[3]: maze.map.getMap()
Out[3]:
[[8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
 [8, 8, 8, 8, 8, 1, 1, 1, 8, 1, 1, 8],
 [8, 1, 8, 8, 8, 8, 1, 1, 8, 1, 1, 8],
 [8, 1, 1, 8, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 0, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 8],
 [8, 8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]]
```

Figure 7 Uniform search with (7, 0) as a wall



```
In[4]: maze.greedySearch()
Greedy search complete.
 Total cost for Uniform Search: 204.815254438
Number of nodes visited: 1552
Path to goal node: [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [4, 6], [3, 7], [2, 8], [1, 8], [0, 9], [0, 10], [1, 11], [2, 11], [3, 11], [4, 11], [5, 11],
number of nodes in path: 40
In[5]: maze.map.getMap()
Out[5]:
[[8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
 [8, 8, 8, 8, 8, 1, 1, 1, 8, 1, 1, 8],
 [8, 1, 8, 8, 8, 8, 1, 1, 8, 1, 1, 8],
 [8, 1, 1, 8, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 0, 0, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 8],
 [8, 8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]]
```

Figure 8 Greedy Search with (7, 0) as a wall



```
In[6]: maze.A_starSearch()
A* search complete.
 Total cost for Uniform Search: 249.117873612
Number of nodes visited: 3039
Path to goal node: [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [4, 5], [4, 6], [3, 7], [2, 8], [1, 8], [0, 9], [0, 10], [1, 11], [2, 11], [3, 11], [4, 11], [5, 11],
number of nodes in path: 40
In[7]: maze.map.getMap()
Out[7]:
[[8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8],
 [8, 8, 8, 8, 8, 1, 1, 1, 8, 1, 1, 8],
 [8, 1, 8, 8, 8, 8, 1, 1, 8, 1, 1, 8],
 [8, 1, 1, 8, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 8, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 8, 8, 8, 8, 1, 1, 8],
 [8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 0, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 1, 8],
 [8, 8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 8],
 [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]]
```

Figure 9 A*Search with (7, 0) as a wall

Discussion

In Both cases, the Uniform search proved to be the worst in terms of efficiencies i.e. the number nodes visited by the search far exceeded the number of nodes visited in the search methods with heuristics. This can be understood by the fact that the Uniform search is a special case of the BFS and traverses the tree is a Breadth-First manner. Without a heuristic, the search will not expand nodes closer to the goal node depth until all nodes in its current depth have been explored. In contrast, the Greedy and A*search uses a heuristic specific to the search problem to search and expand nodes in a depth that is possibly closer to the goal node depth. In this manner, the heuristic is a tool that is able to traverse the search tree in a Depth-First manner at depths that the algorithm expects the goal node to be. The heuristic eliminates the immediate need to traverse the tree in a strictly Breadth-First manner.

In case 2, all the algorithms find the path around the wall towards the goal state. Again, the Greedy Search proves to be the most efficient. We again expect the Uniform Search to be the least efficient, searching through more nodes before finding the goal state.

In terms of completeness, all the algorithms find the goal state. There is no case where any of the search methods should fail as there are no loops in this search problem, expanded states are never added to the search tree again. However, multiple same states can be found in the tree at different depths, this is understood by the fact that the path towards the goal node might not be optimal, and the occurrence of a complex path toward the goal node is always available.

We can investigate the time of the search method by the number of nodes that the method visited to reach the goal node. In all cases, the Greedy search was the fastest, expanding the least amount of nodes before reaching the goal state, while the A* carefully considered step costs in its algorithm – it took slightly longer in order to find a more optimal path to the goal. Therefore, in this case, A* will always find the optimal path to the goal, as we use the admissible Euclidian Distance as a heuristic. Naturally, Uniform search took the most time, evaluating each depth fully before moving toward nodes deeper into the search tree. The uniform search is not necessarily optimal, as it does not use a heuristic. However in this search problem it will be, we expect optimal goals to be the goal nodes that are higher in the search tree. Therefore the uniform search in this case will be optimal.

Conclusion

With the correct heuristic function, the informed searches prove to be powerful in terms of efficiency and optimality. It provides a way to traverse the tree in a “problem specific” manner to intelligently expand only nodes that it expects to be closer to the goal node. We find that in order to create efficient informed searches, one must carefully decide on meaningful functions for $g(n)$ and $h(n)$.