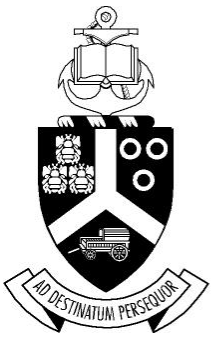

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EAI 320 - Intelligent systems

EAI 320 - Practical Assignment 2 Report

Author:
Mohamed Ameen Omar

Student number:
u16055323

February 22, 2018

Contents

List of Figures

List of Tables

1 Introduction

During the second practical assignment for EAI 320, students were tasked to further their investigation and implementation of different Tree Search strategies. Previously, two Uninformed "Brute Force" search algorithms were investigated and found to be extremely expensive for large data set problems as well as extremely inefficient and thus not suitable for many practical, "real world" problems.

Students have now progressed to the investigation of Informed search strategies, specifically the class of Informed search strategies that take the best first search approach. The two Informed Search strategies that have been implemented and investigated on the given problem scenario are the the Greedy best first search and the A* search.

The Uninformed, Uniform cost search has also been implemented in order to provide a fair comparison between the Informed and Uninformed search strategies.

Both the completeness and the optimality of the different search strategies was to be explored and compared.

2 Background

Informed Search Strategies set themselves apart from Uniformed Search strategies in the fact that, as opposed to Uniformed search strategies, Informed Search strategies use problem specific knowledge beyond the definition of the problem itself, in order to find a solution [1]. This allows for the Informed Search to be more efficient in it's pursuit to find a solution. The Informed Search strategies students will consider in this practical take the best first search approach.

The Greedy best first search makes use of a heuristic function, denoted by $h(n)$, which is dependent on the current state at node n and is the cost estimate for the cheapest path from node n to the goal state.

The A* search is the most widely known best first search [1] and makes use of a cost function that estimates the cheapest or shortest solution through node n denoted by $f(n)$. $f(n)$ is composed of the summation of, the path cost from the start node to node n ($g(n)$) and cost estimate for the cheapest path from node n to the goal state ($h(n)$) [1].

The Uniform cost search, is term as being uninformed and thus does not contain or make use of any information beyond the definition of the problem itself. The search uses a "cost" function, which is usually the distance from one node to it's successor, in order to determine which node to "visit" or evaluate next.

The completeness of a search algorithm is the consistency with which it will return a solution when one exists. A search strategy is termed as being complete if it will always find a solution to a given problem, when one is present. A search strategy is said to be incomplete if in certain situations it will fail to find a solution, even though one is present.

3 Problem Definition

Students were given the Maze shown in figure 1 and were required to design an AI agent that would find a path from the specified starting point (the green block) to the specified end point (the red block).

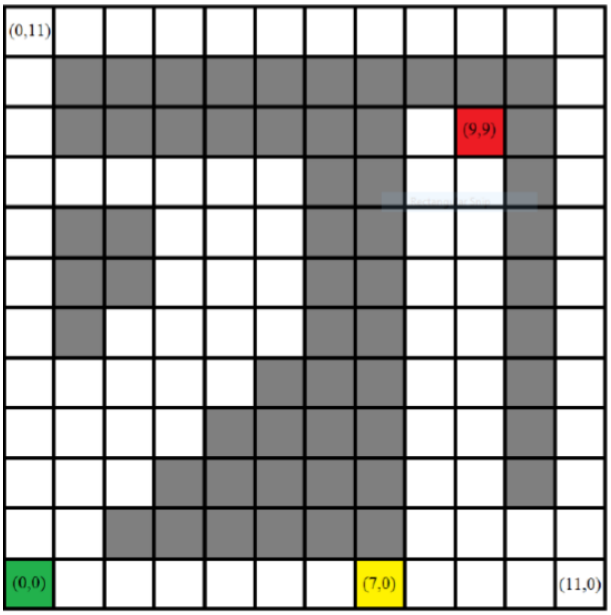

Figure 1: Maze given in the practical specification

The Agent could move horizontally, vertically and diagonally and thus had at most 8 neighbors to which it could traverse. The Grey blocks in figure 1 denoted walls and thus could not be traversed and the yellow block represented an empty space in Task 1 and a wall in Task 2.The problem was to be treated as a Tree search problem, with the root node representing the state for which the agent was located at the starting point and the expansion of the tree was accomplished by the movement of the agent from node to node.

Due to the fact that a block had as much as eight neighbors, this would mean that every node in the tree could have had up to eight neighbors. This presented a problem as the size of the search tree would grow extremely large and a lot of processing would be required in order to first build the tree and thereafter search for the path. In order to overcome this, students were required to dynamically build their search tree as their search algorithms were executed.

The Uniform cost, the Greedy best first and the A* search algorithms needed to be implemented and tailored to the task given. The uniform cost search was to use the Euclidean distance as the "cost" for moving from a node n to one of it's neighboring blocks. The Greedy best first search used the Euclidean distance from the current node to the goal node (9,9) as its heuristic labeled ($h(n)$). The A* star search was to have used the cost function, ($f(n) = g(n) + h(n)$), which is the summation of the total path cost so far and the Euclidean distance from the current node to the goal node.

The yellow block in figure 1 was to represent a normal node, that would be passable in Task 1 and a wall in Task 2.

In order to evaluate the effectiveness of each algorithm and compare them as fairly as possible, a plot of the Maze in figure 1 needed to be printed. The plot needed to be updated by each algorithm illustrating the path found, the walls and the nodes that were expanded or evaluated during the process of finding a result.

4 Implementation and Methodology

My approach in my implementation was to create three separate classes in order to organize my code. In python I created a *node* class, a *myMaze* class and a *game* class, the source for which can be found in Appendix A.

The main purpose of the node class was to represent each position or state in the maze. The class included member variables to keep track of the position, the current path followed by the agent on encountering or moving to a particular state, the cost and the path cost so far. The cost member variable was used to represent the cost or heuristic used by each search algorithm in determining whether the agent would progress to that particular state or to another one of the current state's neighbors. The path cost represented the total cost of moving from the start state to the current state i.e., moving from one state to another vertically or horizontally would incur a cost of 1 and moving to a state that is a diagonal neighbor would incur a cost of$\sqrt{2}$. The class also included several helper functions in order to retrieve information such as whether the node or state was the goal node or state *(isGoal)* and the position of the neighbors of the current state.

The *myMaze* class is used to represent the current state of the Maze as a particular search algorithm executes. It is initialized with a Matrix representing the initial state of the Maze as seen in figure 1. For convenience, a 0 represents a free wall that may be visited or passed and 1 indicates a wall, which is not a valid state to move to. It contains various helper functions such as the reset, which as the name suggests resets the maze to the initial state, various "validate" functions that take in a list of neighbor co ordinates and using the maze, determines whether a certain neighboring state is free, a wall, has been visited or has been expanded, which aids in the implementation of of the three search strategies. Additional helper functions provided functionality for changing the "key" for a particular index in the Maze to indicate if it has been visited or expanded and to make the yellow block in figure 1 a wall or a free node. A *formatMaze* and a *formatMaze* function, are also included. The *formatMaze* function takes in a list which is the path or solution found by a search strategy and the function "formats" the maze so that the output and the status of each node in the maze is clearer. The *printMaze* function is used to output the final Maze plot onto the console for the user.

The game class provides an interface in order to conduct my investigation on the three search strategies. Within this class, I am using a list, *frontier*, that stores nodes that the program has deemed to be a possible node in the solution path and the *getUniExpansionNode*, *getGreedyExpansionNode* and *getStarExpansionNode* functions have been used to iterate through the list, and return the next node to be evaluated based off of the heuristic or cost specific to every type of search. In this way I have made use of my own "pseudo" priority queue, wherein the next node returned from the list, is the node that has the smallest heuristic cost or function. An instance of the *myMaze* class for each search algorithm, for each class is created, along with other member variables in order to keep track of the number of nodes expanded(a node n, whose children have been evaluated to possibly be part of the solution), the number of nodes viewed (node's who have been checked for eligibility to be added to the list of possible solution nodes), the maximum width of the pseudo priority queue that I have opted to use as well as

the path of the solution. The *reset* function is used to restore the current instance of the *game* class to it's original/initialization state. The *uniformSearch*, *greedyBestFirstSearch*, and the *aStarSearch* functions make use of several member "helper" functions to conduct their respective searches.

4.1 The dynamic construction of the search tree and general search procedure

I have taken the search tree approach to the given problem. Due to the fact that each block in the Maze can have as much as 8 neighbors, the branching factor of the tree is extremely high and it would be inefficient to store the information contained in the Maze in a tree first, before conducting a search.

Thus, I am building my tree from the start state to the goal state, but only including the nodes needed to compute the solution. My algorithm begins by first evaluating the start node and adding it's neighbors or successors to a list acting as a pseudo Priority Queue. When these neighbors are added to the list they are marked as being "viewed" and the start node is marked as being "expanded". From the Queue, each search algorithm uses it's respective "helper" function in order to retrieve the next node to be "viewed". This continues until the goal node is reached or an infinite loop is detected. If the goal node is reached, the path to get to the goal node is recorded, the Maze is formatted to show the path, nodes "viewed" and nodes "expanded". Thereafter the Maze and all information needed to evaluate the optimality and completeness of each algorithm is printed to the console.

4.2 Uniform cost search

The Uniform cost search makes use of the cost function $g(n)$ which is the total path cost from the start node to the current node. The node with the smallest value for $g(n)$ is chosen for expansion next. When "viewing" a node's successors during expansion, a check is first done to see if that node has been expanded previously, if it has, this node is not added back to the frontier list. The "goal test" for a node is conducted only when the node is selected for expansion since the first goal node generated may lead to a suboptimal path[1]. A check is also done thereafter to see if the node is currently occupied in the frontier queue, if it is, the two instances of the same node are compared to determine which has the smallest cost function. The instance with the smallest cost function is left/replaced in the queue. The *uniformSearch* function makes use of the *getUniExpansionNode* and the *uniformExpand* "helper" functions in order to conduct the search.

4.3 Greedy best first search

The Greedy best first search only considers nodes that have the shortest cost from the current node to it's neighbor. The keeping track of the number of nodes expanded or

viewed, is handled in the same manner as the Uniform cost search, the only difference is when considering which nodes to add to the frontier queue. During the expansion of each node, the frontier queue is "emptied" and only the neighbors of the node being expanded are added to the queue. All neighbors of a node are added to the queue, whether they have been visited or expanded before, the only nodes being ignored throughout are the nodes marked as a wall. The *greedyBestFirstSearch* function makes use of the *getGreedyExpansionNode* and the *greedyExpand* "helper" functions in order to execute it's search.

4.4 A* Search

The A* search is conducted in the exact same manner as the Uniform cost search, except in the way in the heuristic being used. The summation of the total path cost so far and the cost from the node n to the goal node, is used as the cost heuristic. The *aStarSearch* function makes use of the *starExpand* and the *getStarExpansionNode* "helper" functions in order to complete it's search.

5 Results

The tables below illustrate the details pertaining to the nodes in the dynamic search tree, for each of the search algorithms. Table 1 is when the searches are conducted on the Maze when the block (7,0) is not a wall and Table 2 is for when block (7,0) is a wall. Please note that since the Greedy best first search resulted in an infinite loop that was detected by the search, thus the values in each table cannot be used to draw any conclusion about the optimality or space requirements for the search. They are included purely for completeness.

	Uniform-cost search	**Greedy best first search**	**A* Search**
Total number of nodes in path	17	11**	17
Total path cost	16.83	12.07**	16.83
Total number of nodes expanded	78	11**	55
Total number of nodes viewed	175	51**	143
Maximum number of nodes in search queue	11	7**	13

Table 1: Results for Task 1

	Uniform-cost search	**Greedy best first search**	**A* Search**
Total number of nodes in path	39	11**	39
Total path cost	39.66	12.07**	39.66
Total number of nodes expanded	85	11**	72
Total number of nodes viewed	180	51**	161
Maximum number of nodes in search queue	7	7**	14

Table 2: Results for Task 2

The figures below are screenshots of the console output for each search algorithm.

```
Conducting a Uniform Cost Search
Search Complete
Uniform Search Path Found:
[[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], [8, 8], [9, 9]]
Number of nodes in path 17
Total path cost 16.82842712474619
Number of nodes expanded  78
Number of nodes viewed  175
Maximum number of nodes in search queue:  11
Here is the final Maze:
_____________________________________________________________
|S     2     2     2     2     2     2     2     2     2     2|
|~     2     2     2     2     1     1     1     2     1     1|
|~     1     2     2     2     2     1     1     2     1     1|
|~     1     1     2     2     2     2     2     2     1     1|
|~     1     1     1     2     2     2     2     2     1     1|
|~     1     1     1     1     2     2     2     2     1     1|
|~     1     1     1     1     1     1     1     1     1     1|
|~     1     1     1     1     1     1     1     1     1     3|
|2     ~     ~     ~     ~     ~     ~     ~     ~     2     1|
|2     2     2     2     2     2     2     2     2     E     1|
|2     2     1     1     1     1     1     1     1     1     1|
|2     2     2     2     2     2     2     3     #     #     #|
_____________________________________________________________
Legend
# is free - not expanded or visited
1 is a wall
2 is expanded
3 is viewed or touched
~ is the path
S is the start
E is the end
```

Figure 2: Console output for Uniform cost search conducted for Task 1

```
Conducting a Greedy Best First Search:
Search Complete
The Search resulted in an infinite loop and thus did not find a solution
Greedy Best First Search Followed the following path:
[[0, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [5, 6], [5, 7], [5, 8], [5, 7], [5, 8]]
Number of nodes in path: 11
Number of total path cost: 12.071067811865476
Number of nodes expanded:  11
Number of nodes viewed:  51
Maximum number of nodes in search queue:  7
Here is the final Maze:
```

S	3	3	#	#	#	#	#	#	#	#	#
3	~	3	3	#	1	1	1	#	1	1	#
3	1	~	3	3	#	1	1	#	1	1	#
#	1	1	~	3	3	#	#	#	1	1	#
#	1	1	1	~	3	3	3	3	1	1	#
#	1	1	1	1	~	~	~	~	1	1	#
#	1	1	1	1	1	1	1	1	1	1	#
#	1	1	1	1	1	1	1	1	1	1	#
#	#	#	#	#	#	#	#	#	#	1	#
#	#	#	#	#	#	#	#	#	E	1	#
#	#	1	1	1	1	1	1	1	1	1	#
#	#	#	#	#	#	#	#	#	#	#	#

```
Legend
# is free - not expanded or visited
1 is a wall
2 is expanded
3 is viewed or touched
~ is the path
S is the start
E is the end
```

Figure 3: Console output for Greedy best first search conducted for Task 1

```
Conducting an A* Search
Search Complete
A Star Search Path Found:
[[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], [8, 8], [9, 9]]
Number of nodes in path: 17
Number of total path cost: 16.82842712474619
Number of nodes expanded:  55
Number of nodes viewed:  143
Maximum number of nodes in search queue:  13
Here is the final Maze:
```

S	2	2	2	2	2	2	3	3	#	#
~	2	2	2	2	1	1	2	1	1	#
~	1	2	2	2	2	1	2	1	1	#
~	1	1	2	2	2	2	2	1	1	#
~	1	1	1	2	2	2	2	1	1	#
~	1	1	1	1	2	2	2	1	1	#
~	1	1	1	1	1	1	1	1	1	#
~	1	1	1	1	1	1	1	1	1	#
3	~	~	~	~	~	~	~	3	1	#
3	3	2	2	2	2	2	2	E	1	#
#	3	1	1	1	1	1	1	1	1	#
#	#	#	#	#	#	#	#	#	#	#

```
Legend
# is free - not expanded or visited
1 is a wall
2 is expanded
3 is viewed or touched
~ is the path
S is the start
E is the end
```

Figure 4: Console output for A* search conducted for Task 1

```
Block(7,0) is not passable
Conducting a Uniform Cost Search
Search Complete
Uniform Search Path Found:
[[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], [7, 11], [8, 11], [9, 11], [10, 11],
[11, 10], [11, 9], [11, 8], [11, 7], [11, 6], [11, 5], [11, 4], [11, 3], [11, 2], [10, 1],
[9, 2], [9, 3], [9, 4], [9, 5], [9, 6], [9, 7], [9, 8], [9, 9]]
Number of nodes in path 39
Total path cost 39.65685424949238
Number of nodes expanded  85
Number of nodes viewed  180
Maximum number of nodes in search queue:  7
Here is the final Maze:
```

S	~	~	~	~	~	~	~	~	~	~	2
2	2	2	2	2	1	1	1	2	1	1	~
2	1	2	2	2	2	1	1	2	1	1	~
2	1	1	2	2	2	2	2	2	1	1	~
2	1	1	1	2	2	2	2	2	1	1	~
2	1	1	1	1	2	2	2	2	1	1	~
2	1	1	1	1	1	1	1	1	1	1	~
1	1	1	1	1	1	1	1	1	1	1	~
2	2	2	2	2	2	2	2	2	3	1	~
2	2	~	~	~	~	~	~	~	E	1	~
2	~	1	1	1	1	1	1	1	1	1	~
2	2	~	~	~	~	~	~	~	~	~	2

```
Legend
# is free - not expanded or visited
1 is a wall
2 is expanded
3 is viewed or touched
~ is the path
S is the start
E is the end
```

Figure 5: Console output for Uniform cost search conducted for Task 2

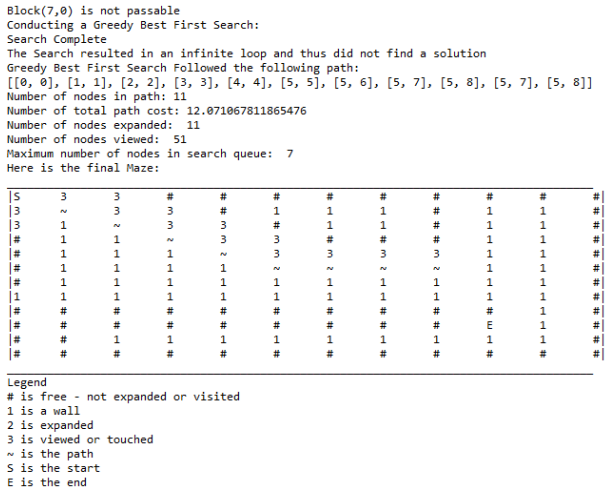

Figure 6: Console output for Greedy best first search conducted for Task 2

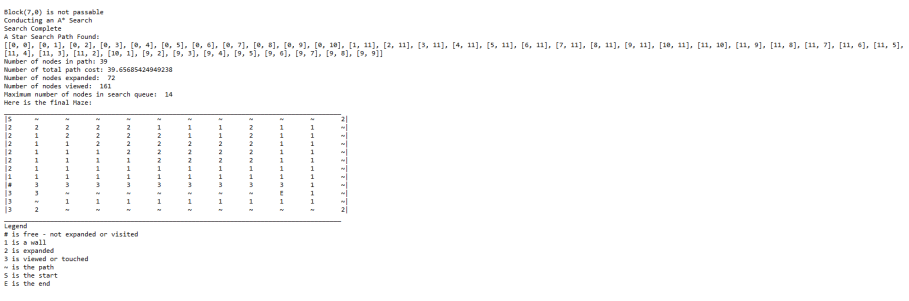

Figure 7: Console output for A* search conducted for Task 2

6 Discussion

The Greedy best first search did not manage to find a result as it encountered an infinite loop between nodes representing blocks (5,7) and (5,8) in both Tasks. This can be seen by referring to figures 3 and 6. This was due to the fact that this scenario was treated as a tree search problem as opposed to a Graph search problem. The Greedy best first search is thus an incomplete search algorithm when used on a search tree. If the Greedy search was done on a graph it would be complete and return a result, since this is a finite space problem (it is incomplete in an infinite space)[1]. The reason for the infinite loop occurring was due to the fact that the algorithm followed a path that lead to a "dead end" and the algorithm does not discriminate nodes that have already been expanded or viewed.

If the search were to find a solution, the efficiency would be greater than that of any other algorithm due to the nature of the Greedy best first search. The search never expands a node that is not on the solution path and therefore does not waste time evaluating unnecessary nodes.This is evident in the unsuccessful search console output in figures 3 and 6, notice that the number of nodes expanded is the same as that of the incorrect path found. The Greedy best first search would also require the least amount of storage, since the number of nodes that would be stored in the frontier queue would at most be the maximum number of neighbors a node may posses. This would become problematic if this number increases drastically as the problem is scaled. Since the storage limit may be exceeded before the search completes at scale. A huge drawback on using a Greedy best first search is the fact that the search is not optimal since it only considers nodes with the shortest distance to the goal node, but does not consider the entire path cost from the start.

The Uniform cost search as well as the A* search managed to find the optimal solution for both the given tasks. The optimal solution was the path: "[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], [8, 8], [9, 9]" with a total path cost of 16.83 and "[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], [7, 11], [8, 11], [9, 11], [10, 11], [11, 10], [11, 9], [11, 8], [11, 7], [11, 6], [11, 5], [11, 4], [11, 3], [11, 2], [10, 1], [9, 2], [9, 3], [9, 4], [9, 5], [9, 6], [9, 7], [9, 8], [9, 9]" with a total path cost of 39.66 for Task 1 and Task 2 respectively. The Uniform cost search will always find the optimal path to the goal(if it is complete), since it always follows the shortest path to a node n, that it expands. However, the optimality of an A* search is dependent on the heuristic function chosen. The heuristic needs to be admissible, meaning the function will never overestimate the cost to reach the end/goal node, in order for the A* search solution to be optimal.

The Uniform cost search is guided by node cost rather than tree depth and considers all valid nodes that have not yet been expanded, thus it is logical to see (and is true) that the search would always return a solution, if one exists on condition that the cost for each state or node in the tree is not zero (it exceeds a small positive constant)[1]. In other words, Uniform cost search is complete under the aforementioned condition. If there are nodes that have a cost of zero, it may lead to the search getting stuck in an

infinite loop. The same can also be said about the A* search, since the only condition required for completeness of an A* search is that the tree contain a finite number of nodes with a heuristic cost less than the total cost of the optimal solution[1].

The Uniform cost search expanded 78 nodes and viewed 175 nodes in order to find the optimal path, whereas the A* search expanded 55 nodes and viewed 143 nodes in Task 1. Clearly the A* search was more efficient than the Uniform cost search, since it required less nodes to be processed and thus less time was needed in order to find the optimal path. The same trend can be seen in the result for Task 2. The number of nodes viewed by each algorithm, is an indication of the number of times a new branch to a node is created in the search tree. The Uniform cost clearly has a larger tree depth and branching factor as compared to A* star. The inefficiency of the Uniform cost search can also be seen in a situation where each node in the tree has an equal cost, since it would compare every leaf in the tree to determine if a leaf exists with a lower cost.

The maximum length of the frontier queue varies between the Uniform cost search and the A* search. A* search does occupy more storage space than Uniform cost, with A* storing double the amount of nodes than Uniform cost. Although A* is more efficient than Uniform cost to find the optimal solution, in terms of time, using an A* search on a problem with a larger state space could be problematic as there is a risk of the storage capacity being reached before the search is complete and thus no result returned. This is not to say that the Uniform cost search does not suffer from the same flaw, however it should be able to handle larger state spaces than A* before running out of storage space.

7 Conclusion

The completeness of a search algorithm is dependent on the type of data structure being searched and the characteristics of the data. As we have observed in the case of the Greedy best first search, if this problem were to be treated as a Graph searching problem, the search would return a solution since the number of nodes are finite, the cost of each node is also finite and the start and end nodes are known. The choice of the heuristic function also plays a major role in the optimality and efficiency of the Informed search algorithms. With the correct choice of heuristic we can guarantee both optimality and efficiency of the A* search (provided certain conditions are met). Although the Uninformed, Uniform cost search would under most "real world" problem conditions provide an optimal result, The Informed searches provide an interface to traverse the tree in a manner that is tailored to each problem. This provides Informed searches with a huge advantage in terms of efficiency, since unnecessary nodes would not be expanded or processed, thus saving a lot of time and processing power (for large scale problems). The A* search is more "intelligent" in the sense that it has an idea of where it is going whereas the Uniform cost search has no knowledge beyond the problem definition.

8 Appendix A: Python Code

```python
#Mohamed Ameen Omar
#u16055323
#EAI 320 - Practical 2
#2018
import math
#represents each postion in the maze
class node:
    def __init__(self,position = [0,0], currentPath = [[0,0]], cost = 0,
    pathCost = 0):
        self.position = position
        self.currentPath = currentPath
        self.cost = cost
        self.pathCost = pathCost

    def addPath(self,path = []):
        self.currentPath = []
        if(len(path) == 0):
            self.currentPath = []
        else:
            for x in range(0, len(path)):
                self.currentPath.append(path[x])

    def isGoal(self):
        if(self.position[0] == 9):
            if(self.position[1] == 9):
                return True
        return False
    #returns a list with the co ordinates of all neighbours
    def getNeighboursPosition(self):
        neighbours = self.getStraightNeighboursPosition()
        neighbours.extend(self.getDiagonalNeighboursPosition())
        return neighbours
```

Figure 8: Program Source Code 1 of 14 [Python]

```python
def getStraightNeighboursPosition(self):
        temp1 = self.position[0]
        temp2 = self.position[1]
        myList = []
        if( ( (temp1-1) >=0) and ((temp1-1) <12)):
            tempList = [temp1-1,temp2]
            myList.append(tempList)
        if( ( (temp1+1) >=0) and ((temp1+1) <12)):
            tempList = [temp1+1,temp2]
            myList.append(tempList)
        if( ( (temp2-1) >=0) and ((temp2-1) <12)):
            tempList = [temp1,temp2-1]
            myList.append(tempList)
        if( ( (temp2+1) >=0) and ((temp2+1) <12)):
            tempList = [temp1,temp2+1]
            myList.append(tempList)
        return myList

    def getDiagonalNeighboursPosition(self):
        temp1 = self.position[0]
        temp2 = self.position[1]
        myList = []
        if( ( (temp1-1) >=0) and ((temp2-1) >=0)):
            tempList = [temp1-1,temp2-1]
            myList.append(tempList)
        if( ( (temp2+1) <12) and ((temp1+1) <12)):
            tempList = [temp1+1,temp2+1]
            myList.append(tempList)
        if( ( (temp2-1) >=0) and ((temp1+1) <12)):
            tempList = [temp1+1,temp2-1]
            myList.append(tempList)
        if( ( (temp1-1) >=0) and ((temp2+1) <12)):
            tempList = [temp1-1,temp2+1]
            myList.append(tempList)
        return myList
```

Figure 9: Program Source Code 2 of 14 [Python]

```
class myMaze:
    def __init__(self):
        self.maze = [[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],
                     [1,1,1,1,1,1,1,1,1,1,1,0],
                     [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]]
        self.maze[7][0] = 0
        # 0 is free
        # 1 is a wall
        # 2 is expanded
        # 3 is viewed or touched
        # 9 is the path

    #returns only free Nodes
    def validateFreeNeighbours(self,neighbours):
        remove = []
        for n in neighbours:
            temp1 = n[0]
            temp2 = n[1]
            if(self.maze[temp1][temp2] != 0):
                remove.append(n)
        for x in remove:
            neighbours.remove(x)
        return neighbours
    #if a NODE passed in is already expanded
    def isExpanded(self,temp):
        x = temp.position[0]
        y = temp.position[1]
        if( self.maze[x][y] == 2):
            return True
        return False
```

Figure 10: Program Source Code 3 of 14 [Python]

```
#returns only free or viewed
    def validateFreeOrViewed(self,neighbours):
        remove = []
        for n in neighbours:
            temp1 = n[0]
            temp2 = n[1]
            if( (self.maze[temp1][temp2] == 1) or (self.maze[temp1][temp2]
    == 2)):
                remove.append(n)
        for x in remove:
             neighbours.remove(x)
        return neighbours

    #returns all except a Wall
    def validateNotWall(self,neighbours):
        remove = []
        for n in neighbours:
            temp1 = n[0]
            temp2 = n[1]
            if( (self.maze[temp1][temp2] == 1)):
                remove.append(n)
        for x in remove:
            neighbours.remove(x)
        return neighbours

    def resetMaze(self):
        self.maze = [[0,0,0,0,0,0,0,0,0,0,0,0,0],
                     [0,0,0,0,0,0,1,1,1,0,1,1,0],
                     [0,1,0,0,0,0,0,1,1,0,1,1,0],
                     [0,1,1,0,0,0,0,0,0,0,1,1,0],
                     [0,1,1,1,0,0,0,0,0,0,1,1,0],
                     [0,1,1,1,1,0,0,0,0,0,1,1,0],
                     [0,1,1,1,1,1,1,1,1,1,1,1,0],
                     [1,1,1,1,1,1,1,1,1,1,1,1,0],
                     [0,0,0,0,0,0,0,0,0,0,0,1,0],
                     [0,0,0,0,0,0,0,0,0,0,0,1,0],
                     [0,0,1,1,1,1,1,1,1,1,1,1,0],
                     [0,0,0,0,0,0,0,0,0,0,0,0,0]]
        self.maze[7][0] = 0
```

Figure 11: Program Source Code 4 of 14 [Python]

```
def toggleNodeExpanded(self,tempNode):
    self.maze[tempNode.position[0]][tempNode.position[1]] = 2

def toggleNodeViewed(self,tempNode):
    self.maze[tempNode.position[0]][tempNode.position[1]] = 3

def makeWall(self):
    self.maze[7][0] = 1
#used to format the maze so that the output is more clear(walls,path,
etc)
#path passed in as an arguement
def formatMaze(self,temp):
    for location in temp:
        self.maze[location[0]][location[1]] = "~" #node in the path
    self.maze[0][0] = "S" #start
    self.maze[9][9] = "E" #end
    for x in range(0,len(self.maze)):
        for y in range(0,len(self.maze[x])):
            if(self.maze[x][y] == 0):#a free node
                self.maze[x][y] = "#"

def printMaze(self):
    for x in range(0,len(self.maze[0])-1):
        print("--------", end = '')
    print('')
    for r in range(0,len(self.maze)):
        print("|", end = '')
        for x in range(0,len(self.maze[r])):
            print(self.maze[r][x], end = '')
            if(x == len(self.maze[r]) -1):
                print("|", end = '')
            print("\t", end = '')
        print()
    for x in range(0,len(self.maze[0])-1):
        print("--------", end = '')
    print()
    print("Legend")
    print("# is free - not expanded or visited\n1 is a wall\n2 is
expanded\n3 is viewed or touched\n~ is the path")
    print("S is the start\nE is the end")
```

Figure 12: Program Source Code 5 of 14 [Python]

```
class game:
    def __init__(self):
        self.start = node() #starting node
        self.uniformSearchMaze = myMaze() #Maze for current instance
        self.uniformNodesExpanded = 0 #number of nodes expanded
        self.uniformNodesViewed = 0 #number of nodes viewed
        self.uniformMaxFrontierQueue = 0; #num of nodes in frontier queue
        self.uniformPath = []
        self.greedySearchMaze = myMaze() #Maze for current instance
        self.greedyNodesExpanded = 0 #number of nodes expanded
        self.greedyNodesViewed = 0 #number of nodes viewed
        self.greedyMaxFrontierQueue = 0;
        self.greedyPath = []
        self.starSearchMaze = myMaze() #Maze for current instance
        self.starNodesExpanded = 0 #number of nodes expanded
        self.starNodesViewed = 0 #number of nodes viewed
        self.starMaxFrontierQueue = 0;
        self.starPath = []
        self.frontier = [] #current nodes that may be expanded

    #resets the game
    def reset(self):
        self.start = node() #starting node
        self.uniformSearchMaze = myMaze() #Maze for current instance
        self.uniformNodesExpanded = 0 #number of nodes expanded
        self.uniformNodesViewed = 0 #number of nodes viewed
        self.uniformMaxFrontierQueue = 0; #num of nodes in frontier queue
        self.uniformPath = []
        self.greedySearchMaze = myMaze() #Maze for current instance
        self.greedyNodesExpanded = 0 #number of nodes expanded
        self.greedyNodesViewed = 0 #number of nodes viewed
        self.greedyMaxFrontierQueue = 0;
        self.greedyPath = []
        self.starSearchMaze = None
        self.starSearchMaze = myMaze() #Maze for current instance
        self.starNodesExpanded = 0 #number of nodes expanded
        self.starNodesViewed = 0 #number of nodes viewed
        self.starMaxFrontierQueue = 0;
        self.starPath = []
        self.frontier = [] #current nodes that may be expanded
```

Figure 13: Program Source Code 6 of 14 [Python]

```
def uniformSearch(self):
        self.frontier = []
        self.frontier.append(self.start)
        temp = None
        self.uniformNodesViewed = 1
        print("Conducting a Uniform Cost Search")
        while(self.frontier != []):
            if(len(self.frontier) > self.uniformMaxFrontierQueue): #to get
    the largest size of queue
                self.uniformMaxFrontierQueue = len(self.frontier)
            temp = self.getUniExpansionNode()#get lowest path cost node
            if(temp.isGoal() == True):
                self.uniformPath = temp.currentPath
                self.uniformSearchMaze.formatMaze(self.uniformPath)
                break
            self.uniformSearchMaze.toggleNodeExpanded(temp)
            self.uniformNodesExpanded += 1
            self.uniformExpand(temp) #expand this node
        print("Search Complete")
        print("Uniform Search Path Found:")
        print(self.uniformPath)
        print("Number of nodes in path", len(self.uniformPath))
        print("Total path cost", temp.pathCost)
        print("Number of nodes expanded ", self.uniformNodesExpanded)
        print("Number of nodes viewed ", self.uniformNodesViewed)
        print("Maximum number of nodes in search queue: ", self.
    uniformMaxFrontierQueue)
        print("Here is the final Maze:")
        self.uniformSearchMaze.printMaze() #print the final maze
        print()
        print()
```

Figure 14: Program Source Code 7 of 14 [Python]

```
#expands the current state
    def uniformExpand(self,tempNode):
        if(tempNode == None):
            return
        neighbours = tempNode.getNeighboursPosition()#get all neighbours
        neighbours = self.uniformSearchMaze.validateFreeOrViewed(
    neighbours)#remove all walls and expanded nodes
        for x in neighbours:
            newNode = node()
            newNode.position = [x[0],x[1]]
            newNode.cost = self.getNeighbourCost(tempNode, newNode.
    position)#from current node to new node
            newNode.pathCost = tempNode.pathCost + newNode.cost# entire
    path cost g(n)
            newNode.addPath(tempNode.currentPath)
            newNode.currentPath.append(newNode.position)
            self.uniformNodesViewed += 1
            self.uniformSearchMaze.toggleNodeViewed(newNode)
            if(self.isInFrontier(x) == True):
                self.uniformSearchMaze.toggleNodeViewed(newNode)
                for x in range(0,len(self.frontier)):
                    if(self.frontier[x].position == newNode.position):
                        if(self.frontier[x].pathCost > newNode.pathCost):
                            self.frontier[x] = newNode
                continue
            self.frontier.append(newNode)

    def isInFrontier(self,position):
        for x in self.frontier:
            if(x.position == position):
                return True
        return False

    #uniform search cost from current node to nextNode
    def getNeighbourCost(self,myNode, nextPostion):
        distance = 0
        if(nextPostion[0] >= myNode.position[0]):
            temp = nextPostion[0] - myNode.position[0]
            distance += (temp*temp)
        else:
            temp = myNode.position[0] - nextPostion[0]
            distance += (temp*temp)

        if(nextPostion[1] >= myNode.position[1]):
            temp = nextPostion[1] - myNode.position[1]
            distance += (temp*temp)
        else:
            temp = myNode.position[1] - nextPostion[1]
            distance += (temp*temp)
        return ( math.sqrt(distance) )
```

Figure 15: Program Source Code 8 of 14 [Python]

```
#returns the node we are going to expand, returns node with smallest cost
    def getUniExpansionNode(self):
        tempNode = None
        for x in range(0, len(self.frontier)):
            if(tempNode == None):
                tempNode = self.frontier[x]
            else:
                if(tempNode.pathCost > self.frontier[x].pathCost):
                    tempNode = self.frontier[x]
        self.frontier.remove(tempNode)
        return tempNode
```

Figure 16: Program Source Code 9 of 14 [Python]

```
def greedyBestFirstSearch(self):
        print("Conducting a Greedy Best First Search:")
        self.frontier = []
        self.head = node()
        self.frontier.append(self.start)
        alreadyExpanded = []
        temp = None
        x = 0
        loop = False
        self.greedyNodesViewed = 1
        while(self.frontier != []):
            if(len(self.frontier) > self.greedyMaxFrontierQueue): #to get
    the largest size of queue
                self.greedyMaxFrontierQueue = len(self.frontier)
            temp = self.getGreedyExpansionNode()
            for y in alreadyExpanded:
                if(y.position == temp.position):
                    x+=1
            if(temp.isGoal() == True):
                self.greedyPath = temp.currentPath
                self.greedySearchMaze.formatMaze(self.greedyPath)
                break
            self.greedySearchMaze.toggleNodeExpanded(temp)
            self.greedyExpand(temp)
            self.greedyNodesExpanded += 1
            if(self.greedySearchMaze.isExpanded(temp) == True):
                alreadyExpanded.append(temp)
            if(x >1): #checking for loop
                loop = True
                self.greedyPath = temp.currentPath
                break
        if(loop == True):
            self.greedySearchMaze.formatMaze(self.greedyPath)
            print("Search Complete")
            print("The Search resulted in an infinite loop and thus did
    not find a solution")
            print("Greedy Best First Search Followed the following path:")
            print(self.greedyPath)
            print("Number of nodes in path:", len(self.greedyPath))
            print("Number of total path cost:", temp.pathCost)
            print("Number of nodes expanded: ", self.greedyNodesExpanded)
            print("Number of nodes viewed: ", self.greedyNodesViewed)
            print("Maximum number of nodes in search queue: ", self.
    greedyMaxFrontierQueue)
            print("Here is the final Maze:")
            self.greedySearchMaze.printMaze() #print the final maze
            print()
            print()
            return
        self.greedySearchMaze.formatMaze(self.greedyPath)
        print("Search Complete")
        print("Greedy Best First Search Path Found:")
        print(self.greedyPath)
```

Figure 17: Program Source Code 10 of 14 [Python]

```
print("Number of nodes in path:", len(self.greedyPath))
        print("Number of total path cost:", temp.pathCost)
        print("Number of nodes expanded: ", self.greedyNodesExpanded)
        print("Number of nodes viewed: ", self.greedyNodesViewed)
        print("Maximum number of nodes in search queue: ", self.
    greedyMaxFrontierQueue)
        print("Here is the final Maze:")
        self.greedySearchMaze.printMaze() #print the final maze
        print()
        print()
    #expands the current state
    def greedyExpand(self,tempNode):
        if(tempNode == None):
            return
        neighbours = tempNode.getNeighboursPosition()
        neighbours = self.greedySearchMaze.validateNotWall(neighbours)
        self.frontier = []
        for x in neighbours:
            newNode = node()
            newNode.position = [x[0],x[1]]
            newNode.cost = self.getGreedyCost(newNode.position)
            newNode.pathCost = tempNode.pathCost + self.getNeighbourCost(
    tempNode,newNode.position)
            newNode.addPath(tempNode.currentPath)
            newNode.currentPath.append(newNode.position)
            self.greedyNodesViewed += 1
            if(self.isInFrontier(x) == True):
                for x in range(0,len(self.frontier)):
                    if(self.frontier[x].position == newNode.position):
                        if(self.frontier[x].cost > newNode.cost):
                            self.frontier[x] = newNode
                continue
            self.greedySearchMaze.toggleNodeViewed(newNode)
            self.frontier.append(newNode)
    def getGreedyExpansionNode(self):
        tempNode = None
        for x in range(0, len(self.frontier)):
            if(tempNode == None):
                tempNode = self.frontier[x]
            else:
                if(tempNode.cost > self.frontier[x].cost):
                    tempNode = self.frontier[x]
        self.frontier.remove(tempNode)
        return tempNode
```

Figure 18: Program Source Code 11 of 14 [Python]

```
# h(n) from current to 9,9
    def getGreedyCost(self,myNode):
        d1 = 9 - myNode[0]
        d2 = 9 - myNode[1]
        d1 = d1*d1
        d2 = d2*d2
        return math.sqrt(d1+d2)
    def aStarSearch(self):
        print("Conducting an A* Search")
        self.frontier = []
        self.start = node()
        self.frontier.append(self.start)
        temp = None
        self.starNodesViewed = 1
        while(self.frontier != []):
            if(len(self.frontier) > self.starMaxFrontierQueue):
                self.starMaxFrontierQueue = len(self.frontier)
            temp = self.getStarExpansionNode()
            if(temp.isGoal() == True):
                self.starPath = temp.currentPath
                self.starSearchMaze.formatMaze(self.starPath)
                break
            self.starSearchMaze.toggleNodeExpanded(temp)
            self.starExpand(temp)
            self.starNodesExpanded += 1
        print("Search Complete")
        print("A Star Search Path Found:")
        print(self.starPath)
        print("Number of nodes in path:", len(self.starPath))
        print("Number of total path cost:", temp.pathCost)
        print("Number of nodes expanded: ", self.starNodesExpanded)
        print("Number of nodes viewed: ", self.starNodesViewed)
        print("Maximum number of nodes in search queue: ", self.
    starMaxFrontierQueue)
        print("Here is the final Maze:")
        self.starSearchMaze.printMaze() #print the final maze
        print()
        print()
```

Figure 19: Program Source Code 12 of 14 [Python]

```
#expands the current state
    def starExpand(self,tempNode):
        if(tempNode == None):
            return
        neighbours = tempNode.getNeighboursPosition()
        neighbours = self.starSearchMaze.validateFreeOrViewed(neighbours)
        for x in neighbours:
            newNode = node()
            newNode.position = [x[0],x[1]]
            newNode.pathCost = tempNode.pathCost + self.getNeighbourCost(
    tempNode,newNode.position)
            newNode.cost = self.getGreedyCost(newNode.position) + newNode.
    pathCost #g+h
            newNode.addPath(tempNode.currentPath)
            newNode.currentPath.append(newNode.position)
            self.starNodesViewed += 1
            if(self.isInFrontier(x) == True):
                for x in range(0,len(self.frontier)):
                    if(self.frontier[x].position == newNode.position):
                        if(self.frontier[x].cost > newNode.cost):
                            self.frontier[x] = newNode
                continue
            self.starSearchMaze.toggleNodeViewed(newNode)
            self.frontier.append(newNode)

    #return next node from list for A * Search
    def getStarExpansionNode(self):
        tempNode = None
        for x in range(0, len(self.frontier)):
            if(tempNode == None):
                tempNode = self.frontier[x]
            else:
                if(tempNode.cost > self.frontier[x].cost):#g+h
                    tempNode = self.frontier[x]

        returnNode = tempNode
        self.frontier.remove(tempNode)
        return returnNode
```

Figure 20: Program Source Code 13 of 14 [Python]

```
def wallUniform(self):
        self.reset()
        print("Block(7,0) is not passable")
        self.uniformSearchMaze.makeWall()
        self.uniformSearch()

    def wallGreedy(self):
        self.reset()
        print("Block(7,0) is not passable")
        self.greedySearchMaze.makeWall()
        self.greedyBestFirstSearch()

    def wallStar(self):
        self.reset()
        print("Block(7,0) is not passable")
        self.starSearchMaze.makeWall()
        self.aStarSearch()

newGame = game()
newGame.uniformSearch()
newGame.greedyBestFirstSearch()
newGame.aStarSearch()
newGame.wallUniform()
newGame.wallGreedy()
newGame.wallStar()
```

Figure 21: Program Source Code 14 of 14 [Python]

9 Bibliography

[1] S. Russel and P. Norvig, *Artificial intelligence : A modern approach*, Third. Pearson, 2010.