Queen’s University

Graph Pathfinding and Alpha-Beta Pruning Algorithms

CISC 352

Assignment 2

Professor Sidney Givigi

March 13, 2020

Group 16

Christina Hu	[20093642]
Jie Niu	[20072214]
Wansi Liang	[20067725]
Gary Wang	[20073161]
Derek Huang	[20022672]

Purpose

To find the shortest path to reach the goal from the starting point, the Greedy algorithm and A* algorithm are both used to be implemented. To grab the score and the number of examined leaf nodes of a given input graph, the alpha-beta pruning algorithm is implemented for solving it.

Introduction

Pathfinding

Pathfinding is the plotting of the shortest route between two points. This field of research is based heavily on Dijkstra's algorithm for finding a shortest path on a weighted graph. At its core, a pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node (goal) is reached, generally with the intent of finding the shortest route. A core use for pathfinding in artificial intelligence is for determining routes for agents to follow.

Several algorithms exist for pathfinding. Dijkstra's algorithm (aka uniform-cost) was developed in 1959 by Edsger Dijkstra. It follows the cheapest path cost and is optimal, but it is inefficient when compared to other pathfinding algorithms. Greedy Best-First Search (or just Greedy Search) was described by Judea Pearl. It follows the cheapest heuristic cost and is generally efficient, but it is not optimal. Peter Hart, Nils Nilsson, and Bertram Raphael of Stanford Research Institute first described the A* algorithm in 1968 as an extension of Dijkstra's algorithm. A* is both optimal and efficient. It takes into consideration both the path cost and the heuristic cost.

<u>Alpha-Beta Pruning</u>

A minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player tame. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is A's turn to move, A gives a value to each of his legal moves.

Alpha-beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated sooner. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Methods and Processes

<u>Pathfinding</u>

The python script `pathfinding.py` takes input from *"pathfinding_a.txt"* or *"pathfinding_b.txt"* and output to "*pathfinding_a_out.txt*" or *"pathfinding_b_out.txt"*.

To run the Pathfinding program, make sure that *pathfinding_a.txt* file is in the same folder. This text file contains multiple lines of input. The input should consist of several m x n grids, each separated by a blank line between them. However, the input will not contain the size of the grid.

1. The class point defines a point in the maze problem, where it has the location, cost and origin they come from in relation to where it is stored. So it defines a point to be consisting of its x orientation, y orientation, where it came from, as well as the cost so far in response to the given point.

2. Function *pointfinder()* is a function which is used to find the location of the start of the goal. It takes two arguments, which are namely our current self object, as well as the target goal that the algorithm is aiming to reach.

3. Function *greedySearchAlgorithm()* is a function which is a Greedy algorithm to solve the maze problem. It may allow or not allow to move diagonally, and this is specified in the second parameter. So, this function takes two arguments, which are namely our current self object, as well as the diagonal second boolean argument that specifies whether we can move diagonally or not, with the default value to be set as false.

4. Function *AStarSearchAlgorithm()* is a function which is an A* algorithm to solve the maze problem. As pretty much similar as above, it may allow or not allow to move diagonally, and this is specified in the second parameter. So, this function takes two arguments, which are namely our current self object, as well as the diagonal second boolean argument that specifies whether we can move diagonally or not, with the default value to be set as false.

5. Function *getXYNeighbours()* is a function which is used to find the neighbours of a given point, both horizontally and vertically. So this function right here takes in two arguments, which are namely our current self object, as well as the second argument that specifies a point where the algorithm will be computing this point's neighbors.

6. Function *getDiagNeighbours()* is a function which is used to find the neighbours of a point diagonally. So this function is pretty similar to the above function, except that it

finds only the diagonal neighbors. This function takes in two arguments, which are namely our current self object, as well as the second argument that specifies a point where the algorithm will be computing this point's diagonal neighbors.

7. Function *chebyshev()* is a heuristic function which is used to find the chebyshev distance from point a to point b. This function takes in three arguments, which are namely our current self object, as well as the second argument which is the starting point a and the third argument which is the ending point b. It returns the calculated chebyshev distance from point a to point b.

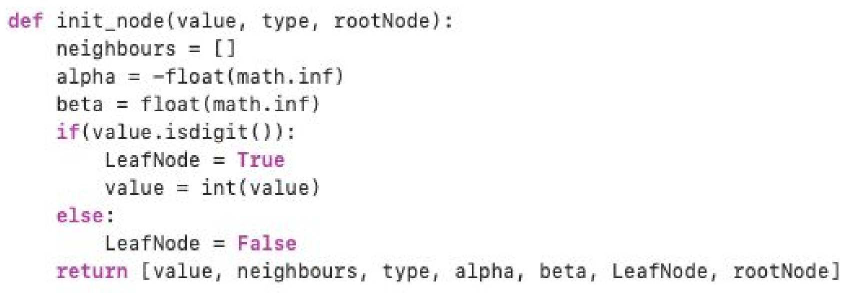
8. Function *manhattan()* is a heuristic function which is used to find the manhattan distance from point a to point b. So this function is pretty similar to the above function, except that it uses manhattan instead of chebyshev as the distance style. This function takes in three arguments, which are namely our current self object, as well as the second argument which is the starting point a and the third argument which is the ending point b. It returns the calculated manhattan distance from point a to point b.

<u>Alpha-beta pruning</u>

The python script `alphabeta.py` takes input from *"alphabeta.txt"* and output to "*alphabeta_out.txt*".

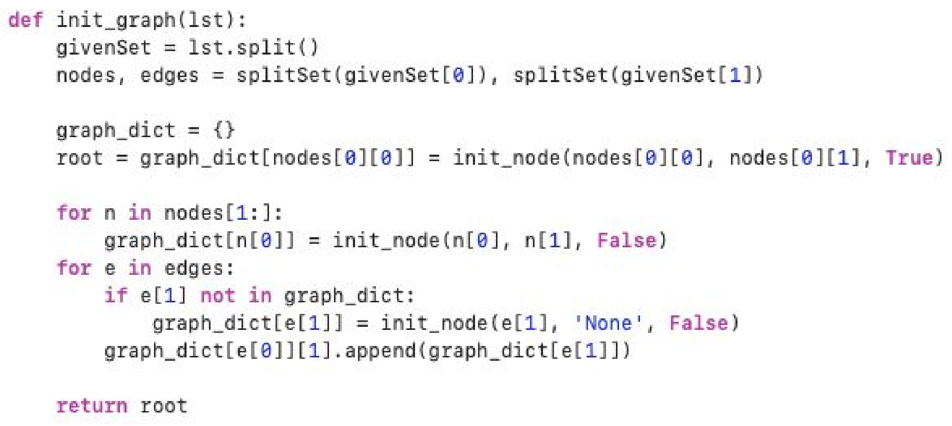
To run the Alpha-Beta Pruning program, make sure that *alphabeta.txt* file is in the same folder. This text file contains multiple lines of input. The input "*alphabeta.txt*" should contain multiple graphs to be examined. These will be in the form of a set of nodes along with its type ("MAX" or "MIN") followed by a space and then a set of edges between the nodes.

1. Function *init_node()* takes input of the value, the type of the node, and a boolean value of whether the current node object is the root node of a graph. It returns the properties of the node object including its value, neighbours, type, the numerical value of alpha and beta, and boolean expression of whether the node is a leaf node or a root node.



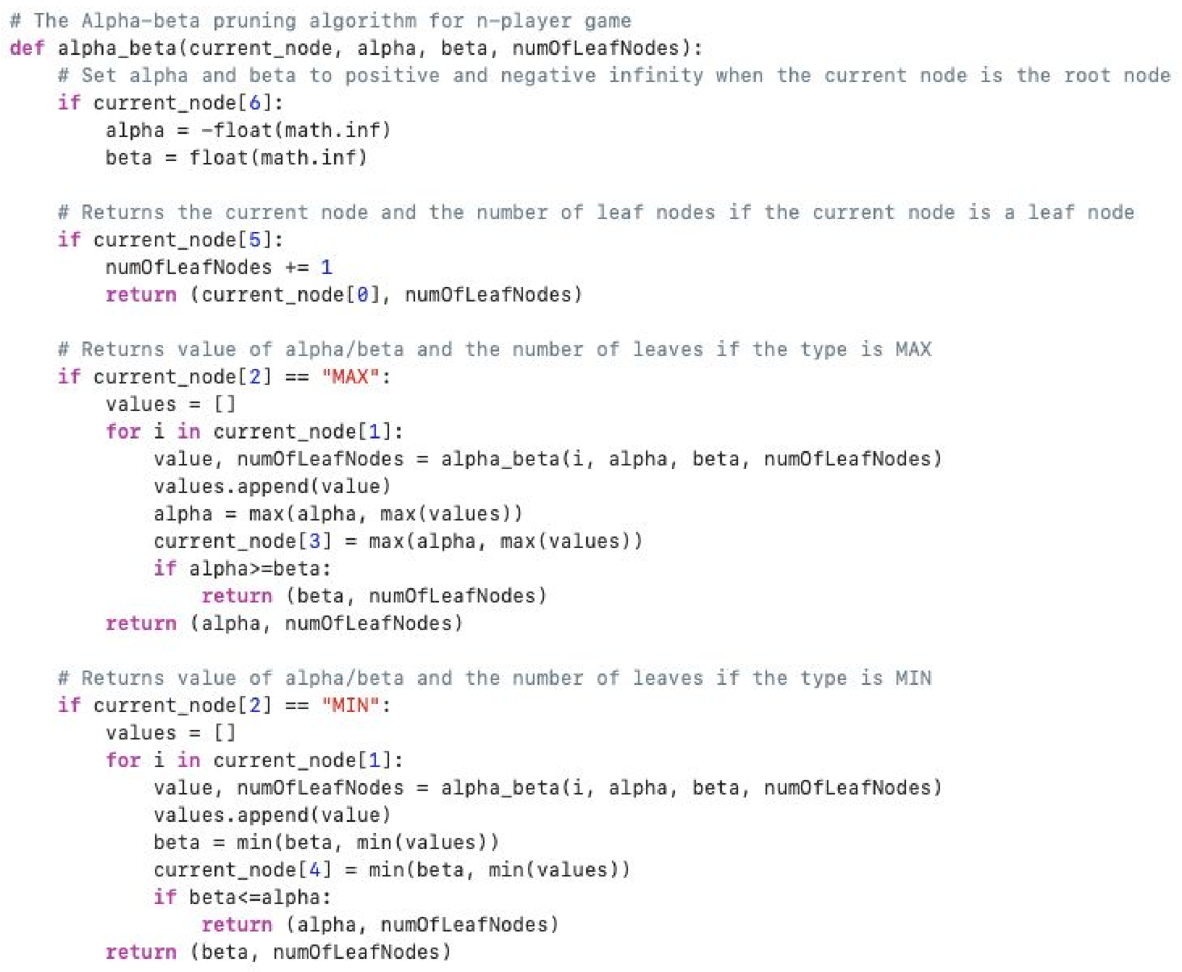
```
def init_node(value, type, rootNode):
    neighbours = []
    alpha = -float(math.inf)
    beta = float(math.inf)
    if(value.isdigit()):
        LeafNode = True
        value = int(value)
    else:
        LeafNode = False
    return [value, neighbours, type, alpha, beta, LeafNode, rootNode]
```

2. Function *init_graph()* takes a list-like string as the parameter. It constructs a graph structure with nodes initialized. The root node of the graph is returned to the function.



```
def init_graph(lst):
    givenSet = lst.split()
    nodes, edges = splitSet(givenSet[0]), splitSet(givenSet[1])

    graph_dict = {}
    root = graph_dict[nodes[0][0]] = init_node(nodes[0][0], nodes[0][1], True)

    for n in nodes[1:]:
        graph_dict[n[0]] = init_node(n[0], n[1], False)
    for e in edges:
        if e[1] not in graph_dict:
            graph_dict[e[1]] = init_node(e[1], 'None', False)
        graph_dict[e[0]][1].append(graph_dict[e[1]])

    return root
```

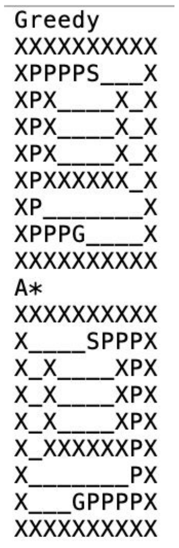
3. Function *alpha_beta()* is a recursive algorithm that takes the current node object, the value of alpha and beta, and the number of leaf nodes as parameters. The function checks if the node is either a root or a leaf. Then it checks the type of the node and recursively runs this algorithm for all leaf nodes. Finally, it returns the score from alpha or beta and the number of leaf nodes examined.



```python
# The Alpha-beta pruning algorithm for n-player game
def alpha_beta(current_node, alpha, beta, numOfLeafNodes):
    # Set alpha and beta to positive and negative infinity when the current node is the root node
    if current_node[6]:
        alpha = -float(math.inf)
        beta = float(math.inf)

    # Returns the current node and the number of leaf nodes if the current node is a leaf node
    if current_node[5]:
        numOfLeafNodes += 1
        return (current_node[0], numOfLeafNodes)

    # Returns value of alpha/beta and the number of leaves if the type is MAX
    if current_node[2] == "MAX":
        values = []
        for i in current_node[1]:
            value, numOfLeafNodes = alpha_beta(i, alpha, beta, numOfLeafNodes)
            values.append(value)
            alpha = max(alpha, max(values))
            current_node[3] = max(alpha, max(values))
            if alpha>=beta:
                return (beta, numOfLeafNodes)
        return (alpha, numOfLeafNodes)

    # Returns value of alpha/beta and the number of leaves if the type is MIN
    if current_node[2] == "MIN":
        values = []
        for i in current_node[1]:
            value, numOfLeafNodes = alpha_beta(i, alpha, beta, numOfLeafNodes)
            values.append(value)
            beta = min(beta, min(values))
            current_node[4] = min(beta, min(values))
            if beta<=alpha:
                return (alpha, numOfLeafNodes)
        return (beta, numOfLeafNodes)
```

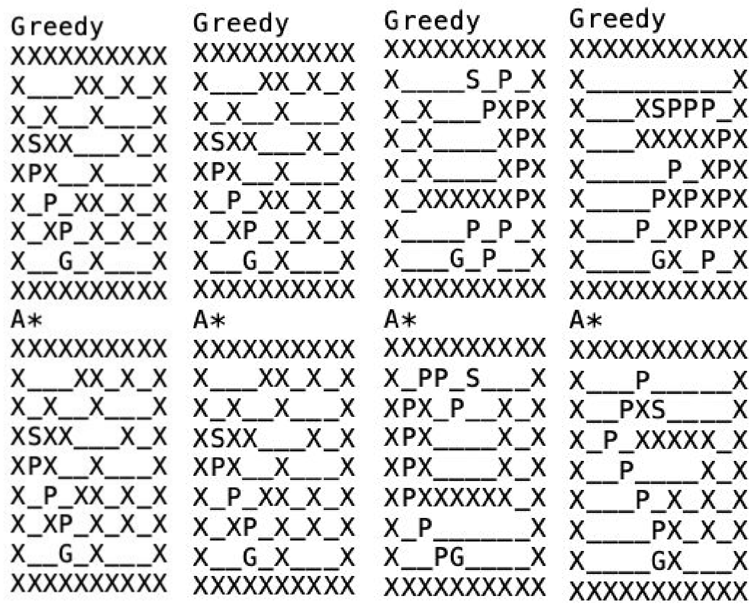
Results

Pathfinding (Part1)



```
Greedy
XXXXXXXXXX
XPPPPS___X
XPX____X_X
XPX____X_X
XPX____X_X
XPXXXXXX_X
XP_______X
XPPPG____X
XXXXXXXXXX
A*
XXXXXXXXXX
X____SPPPX
X_X____XPX
X_X____XPX
X_X____XPX
X_XXXXXXPX
X_______PX
X___GPPPPX
XXXXXXXXXX
```

Figure 1: output of sample input file pathfinding_a.txt



```
Greedy
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
XPX__X___X
X_P_XX_X_X
X_XP_X_X_X
X__G_X___X
XXXXXXXXXX
A*
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
XPX__X___X
X_P_XX_X_X
X_XP_X_X_X
X__G_X___X
XXXXXXXXXX
```

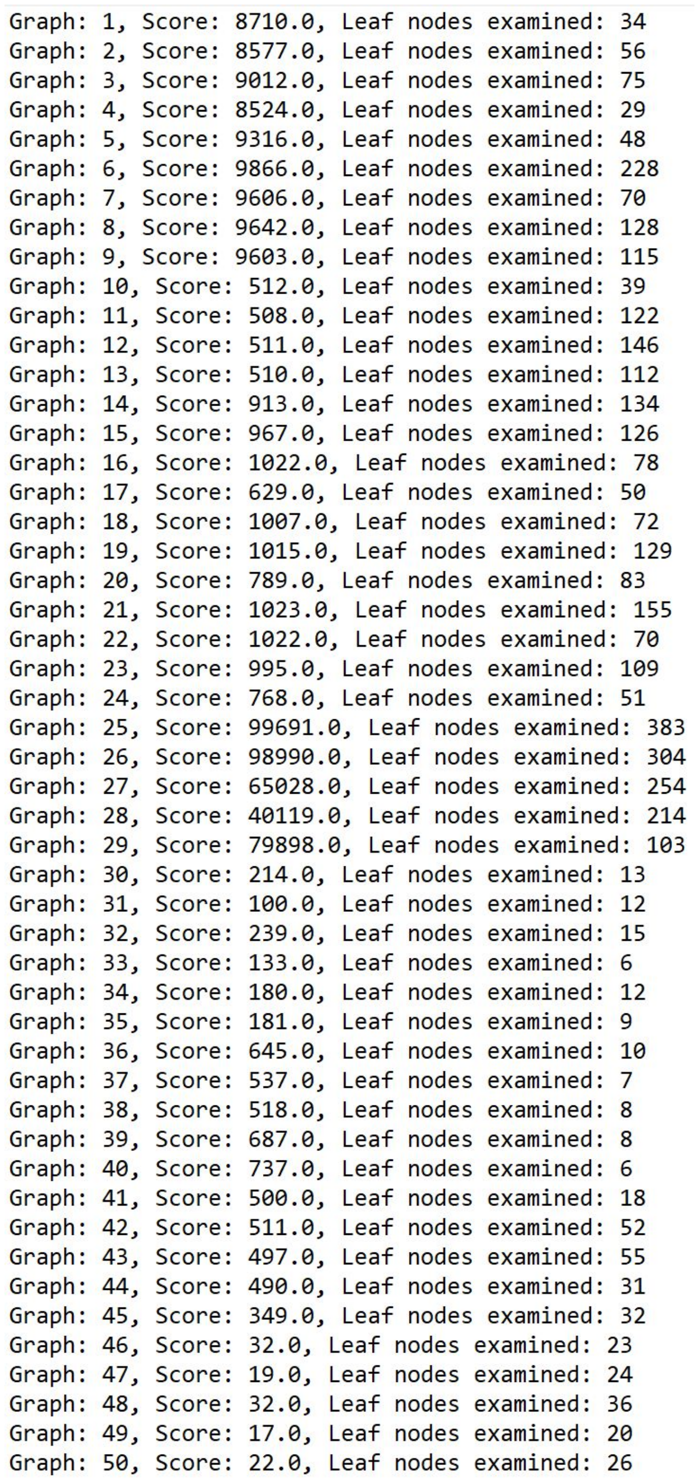
```
Greedy
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
XPX__X___X
X_P_XX_X_X
X_XP_X_X_X
X__G_X___X
XXXXXXXXXX
A*
XXXXXXXXXX
X___XX_X_X
X_X__X___X
XSXX___X_X
XPX__X___X
X_P_XX_X_X
X_XP_X_X_X
X__G_X___X
XXXXXXXXXX
```

```
Greedy
XXXXXXXXXX
X____S_P_X
X_X___PXPX
X_X____XPX
X_X____XPX
X_XXXXXXPX
X____P_P_X
X___G_P__X
XXXXXXXXXX
A*
XXXXXXXXXX
X_PP_S___X
XPX_P__X_X
XPX____X_X
XPX____X_X
XPXXXXXX_X
X_P______X
X__PG____X
XXXXXXXXXX
```

```
Greedy
XXXXXXXXXXX
X_________X
X___XSPPP_X
X___XXXXXPX
X_____P_XPX
X____PXPXPX
X___P_XPXPX
X____GX_P_X
XXXXXXXXXXX
A*
XXXXXXXXXXX
X___P_____X
X__PXS____X
X_P_XXXXX_X
X__P____X_X
X___P_X_X_X
X____PX_X_X
X____GX___X
XXXXXXXXXXX
```

Figure 2: outputs of sample input file pathfinding_b.txt

Alpha-beta pruning (Part2)



```
Graph: 1, Score: 8710.0, Leaf nodes examined: 34
Graph: 2, Score: 8577.0, Leaf nodes examined: 56
Graph: 3, Score: 9012.0, Leaf nodes examined: 75
Graph: 4, Score: 8524.0, Leaf nodes examined: 29
Graph: 5, Score: 9316.0, Leaf nodes examined: 48
Graph: 6, Score: 9866.0, Leaf nodes examined: 228
Graph: 7, Score: 9606.0, Leaf nodes examined: 70
Graph: 8, Score: 9642.0, Leaf nodes examined: 128
Graph: 9, Score: 9603.0, Leaf nodes examined: 115
Graph: 10, Score: 512.0, Leaf nodes examined: 39
Graph: 11, Score: 508.0, Leaf nodes examined: 122
Graph: 12, Score: 511.0, Leaf nodes examined: 146
Graph: 13, Score: 510.0, Leaf nodes examined: 112
Graph: 14, Score: 913.0, Leaf nodes examined: 134
Graph: 15, Score: 967.0, Leaf nodes examined: 126
Graph: 16, Score: 1022.0, Leaf nodes examined: 78
Graph: 17, Score: 629.0, Leaf nodes examined: 50
Graph: 18, Score: 1007.0, Leaf nodes examined: 72
Graph: 19, Score: 1015.0, Leaf nodes examined: 129
Graph: 20, Score: 789.0, Leaf nodes examined: 83
Graph: 21, Score: 1023.0, Leaf nodes examined: 155
Graph: 22, Score: 1022.0, Leaf nodes examined: 70
Graph: 23, Score: 995.0, Leaf nodes examined: 109
Graph: 24, Score: 768.0, Leaf nodes examined: 51
Graph: 25, Score: 99691.0, Leaf nodes examined: 383
Graph: 26, Score: 98990.0, Leaf nodes examined: 304
Graph: 27, Score: 65028.0, Leaf nodes examined: 254
Graph: 28, Score: 40119.0, Leaf nodes examined: 214
Graph: 29, Score: 79898.0, Leaf nodes examined: 103
Graph: 30, Score: 214.0, Leaf nodes examined: 13
Graph: 31, Score: 100.0, Leaf nodes examined: 12
Graph: 32, Score: 239.0, Leaf nodes examined: 15
Graph: 33, Score: 133.0, Leaf nodes examined: 6
Graph: 34, Score: 180.0, Leaf nodes examined: 12
Graph: 35, Score: 181.0, Leaf nodes examined: 9
Graph: 36, Score: 645.0, Leaf nodes examined: 10
Graph: 37, Score: 537.0, Leaf nodes examined: 7
Graph: 38, Score: 518.0, Leaf nodes examined: 8
Graph: 39, Score: 687.0, Leaf nodes examined: 8
Graph: 40, Score: 737.0, Leaf nodes examined: 6
Graph: 41, Score: 500.0, Leaf nodes examined: 18
Graph: 42, Score: 511.0, Leaf nodes examined: 52
Graph: 43, Score: 497.0, Leaf nodes examined: 55
Graph: 44, Score: 490.0, Leaf nodes examined: 31
Graph: 45, Score: 349.0, Leaf nodes examined: 32
Graph: 46, Score: 32.0, Leaf nodes examined: 23
Graph: 47, Score: 19.0, Leaf nodes examined: 24
Graph: 48, Score: 32.0, Leaf nodes examined: 36
Graph: 49, Score: 17.0, Leaf nodes examined: 20
Graph: 50, Score: 22.0, Leaf nodes examined: 26
```

Figure 3: outputs of sample input file alphabeta.txt

Conclusion

In summary, the Greedy algorithm and A* algorithm are both very efficient ways to find the shortest path with the lowest total weight from two points. Looking at their respective final solutions, we can observe that it may not always be the case that either algorithm will come up with the same solution, but it is interesting to note that both algorithms solve the problem very well and that one might be more efficient than the other or vice versa. It successfully helps us to find the shortest path given an input grid, which contributes to simulate an intelligent agent in the field of Artificial Intelligence.

On the other hand, the implementation of Alpha-Beta Pruning algorithm helps us to successfully determine the score and number of leaf nodes examined on a given input graph. Looking at its output, we can observe that Alpha-Beta Pruning algorithm is an efficient algorithm that helps us quickly calculate the score with examining the least number of nodes. Hence, the Alpha-Beta Pruning algorithm is also a successful agent contributed to the field of Artificial Intelligence.

References

S. Givigi. (2020). Assignment 2, Artificial Intelligence, CISC 352, Winter, 2020.