# Question 1

For all parts of Question 1, repeated states are ignored as per the assignment instructions, therefore optimality is no longer guaranteed.

1. **Uniform Cost Search**

Form: ng(n), where n is the city number, and g(n) is the actual travel distance from city 1 to city n.

|  |  |  |
| --- | --- | --- |
| **Expanded Node** | **Open Queue** | **Closed Queue** |
|  | { 10 } |  |
| 10 | { 55 824 } |  |
| 55 | { 824 640 } | { 10 } |
| 824 | { 1039 640 347 } | { 10 55 } |
| 1039 | { 640 347 965 } | { 10 55 824 } |
| 640 | { 347 965 278 } | { 10 55 824 1039 } |
| 347 | { 454 965 278 } | { 10 55 824 1039 640 } |
| 454 | { 965 278 } | { 10 55 824 1039 640 347 } |
| 965 | { 278 7100 } | { 10 55 824 1039 640 347 454 } |
| 278 | { 7100 } | { 10 55 824 1039 640 347 454 965 } |
| 7100 | { } | { 10 55 824 1039 640 347 454 965 278 } |

Path: 1 🡪 8 🡪 10 🡪 9 🡪 7

Cost: 100

1. **Greedy Best First Search**

Form: nh(n), where n is the city number, and h(n) is an estimate of distance from city n to city 7.

|  |  |  |
| --- | --- | --- |
| **Expanded Node** | **Open Queue** | **Closed Queue** |
|  | { 178 } |  |
| 178 | { 860 575 } |  |
| 860 | { 337 1057 575 } | { 178 } |
| 337 | { 430 1057 575 } | { 178 860 } |
| 430 | { 935 1057 575 } | { 178 860 337 } |
| 935 | { 70 232 1057 660 575 } | { 178 860 337 430 } |
| 70 | { 232 1057 660 575 } | { 178 860 337 430 935 } |

Path: 1 🡪 8 🡪 3 🡪 4 🡪 9 🡪 7

Cost: 107

1. **A\* Search**

Form: ng(n)+h(n), where n is the city number, g(n) is the actual travel distance from city 1 to city n, and h(n) is an estimate of distance from city n to city 7.

|  |  |  |
| --- | --- | --- |
| **Expanded Node** | **Open Queue** | **Closed Queue** |
|  | { 178 } |  |
| 178 | { 580 884 } |  |
| 580 | { 884 6100 } | { 178 } |
| 884 | { 384 1096 6100 } | { 178 580 } |
| 384 | { 484 1096 6100 } | { 178 580 884 } |
| 484 | { 1096 6100 9107 } | { 178 580 884 384 } |
| 1096 | { 6100 9100 9107 } | { 178 580 884 384 484 } |
| 6100 | { 9100 9101 9107 2110 } | { 178 580 884 384 484 1096 } |
| 9100 | { 7100 9101 9107 2110 2123 } | { 178 580 884 384 484 1096 6100 } |
| 7100 | { 2110 2123 } | { 178 580 884 384 484 1096 6100 9100 } |

Path: 1 🡪 8 🡪 10 🡪 9 🡪 7

Cost: 100

# Question 2

To simplify setup of the maze in my program, let A represent E1 and B represent E2. This is to allow the maze to be structured as a 2D vector of characters as opposed to strings, saving on memory requirements and simplifying usage. So, in the code A and B are used, but in this assignment document E1 and E2 are used. Additionally, I assumed (0, 0) to be the top left of the maze, and (24, 24) be the bottom right. This simplifies coding since this is naturally how 2D vectors are indexed. So, (24, 0) and (0, 24) is used in the code to represent bottom left and top right, but (0, 0) and (24, 24) is used in the document.

1. **Breadth First Search**

Breadth first search was implemented using a queue data structure, a 2D vector to track visited nodes, and a 2D vector containing adjacent directions of search. While the queue is not empty, pop the front of the queue and check if it is equal to the destination. If so return, but if not, iterate through the adjacent directions. For each adjacent direction, if that node is valid, meaning within bounds, not yet visited, and is not a wall, then push it to the queue and count up the number of nodes explored. Additionally, have a map that tracks the parent nodes with all its children nodes, with the keys being the parent nodes, and the values being its list of children nodes. Once the destination is found, trace its complete path using the map, count up its cost, and return these along with the nodes explored.

|  |
| --- |
| **Starting at S and ending at E1** |
| Path: (13, 2), (13, 3), (12, 3), (12, 4), (12, 5), (12, 6), (12, 7), (11, 7), (10, 7), (9, 7), (9, 8), (9, 9), (8, 9), (7, 9), (7, 10), (7, 11), (7, 12), (7, 13), (7, 14), (7, 15), (7, 16), (7, 17), (7, 18), (7, 19), (7, 20), (7, 21), (7, 22), (7, 23), (6, 23), (5, 23)  Cost: 522  Nodes: 380 |

|  |
| --- |
| **Starting at S and ending at E2** |
| Path: (13, 2), (13, 3), (12, 3), (12, 4), (12, 5), (12, 6), (12, 7), (11, 7), (10, 7), (9, 7), (8, 7), (7, 7), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (6, 2), (5, 2), (4, 2), (3, 2)  Cost: 49  Nodes: 278 |

|  |
| --- |
| **Starting at (0, 0) and ending at (24, 24)** |
| Path: (24, 0), (24, 1), (24, 2), (24, 3), (24, 4), (24, 5), (24, 6), (24, 7), (24, 8), (24, 9), (24, 10), (24, 11), (24, 12), (24, 13), (24, 14), (24, 15), (24, 16), (24, 17), (23, 17), (22, 17), (21, 17), (20, 17), (19, 17), (18, 17), (17, 17), (16, 17), (15, 17), (14, 17), (13, 17), (12, 17), (11, 17), (11, 18), (11, 19), (11, 20), (10, 20), (10, 21), (9, 21), (9, 22), (8, 22), (8, 23), (7, 23), (7, 24), (6, 24), (5, 24), (4, 24), (3, 24), (2, 24), (1, 24), (0, 24)  Cost: 77  Nodes: 447 |

1. **Depth First Search**

Depth first search was implemented using the call stack, a 2D vector to track visited nodes, and a 2D vector to store the path. A recursive approach was taken where the call stack was used to search in a last in, first out manner. If the next node to be searched has not yet been visited, set it to visited, push this coordinate to the path, and update the cost and node counters. The base case is if the destination has been reached, then return true and output the complete path, cost, and number of nodes explored. The recursive case is to search all the way left, then up, then right, then down, while ensuring the moves are valid, meaning within bounds and not a wall. If it passes all these recursive cases within the call stack, then set visited to false, pop from the path, and decrease the cost and nodes explored.

|  |
| --- |
| **Starting at S and ending at E1** |
| Path: (13, 2), (13, 1), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (7, 9), (7, 10), (7, 11), (6, 11), (5, 11), (4, 11), (3, 11), (3, 10), (3, 9), (2, 9), (2, 10), (1, 10), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (1, 16), (1, 15), (1, 14), (1, 13), (1, 12), (1, 11), (2, 11), (2, 12), (2, 13), (3, 13), (3, 12), (4, 12), (4, 13), (5, 13), (5, 12), (6, 12), (6, 13), (7, 13), (7, 12), (8, 12), (8, 11), (9, 11), (9, 12), (9, 13), (8, 13), (8, 14), (7, 14), (7, 15), (7, 16), (7, 17), (7, 18), (7, 19), (7, 20), (7, 21), (6, 21), (5, 21), (4, 21), (3, 21), (2, 21), (1, 21), (1, 20), (1, 19), (0, 19), (0, 20), (0, 21), (0, 22), (0, 23), (0, 24), (1, 24), (1, 23), (1, 22), (2, 22), (2, 23), (2, 24), (3, 24), (3, 23), (3, 22), (4, 22), (4, 23), (4, 24), (5, 24), (5, 23)  Cost: 707  Nodes: 98 |

|  |
| --- |
| **Starting at S and ending at E2** |
| Path: (13, 2), (13, 1), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (7, 7), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (6, 2), (6, 1), (6, 0), (5, 0), (4, 0), (3, 0), (3, 1), (3, 2)  Cost: 203  Nodes: 29 |

|  |
| --- |
| **Starting at (0, 0) and ending at (24, 24)** |
| Path: (24, 0), (23, 0), (22, 0), (21, 0), (21, 1), (21, 2), (21, 3), (20, 3), (19, 3), (18, 3), (18, 4), (18, 5), (17, 5), (16, 5), (15, 5), (15, 4), (15, 3), (15, 2), (15, 1), (15, 0), (14, 0), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (7, 9), (7, 10), (7, 11), (6, 11), (5, 11), (4, 11), (3, 11), (3, 10), (3, 9), (2, 9), (2, 10), (1, 10), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (1, 16), (1, 15), (1, 14), (1, 13), (1, 12), (1, 11), (2, 11), (2, 12), (2, 13), (3, 13), (3, 12), (4, 12), (4, 13), (5, 13), (5, 12), (6, 12), (6, 13), (7, 13), (7, 12), (8, 12), (8, 11), (9, 11), (9, 12), (9, 13), (8, 13), (8, 14), (7, 14), (7, 15), (7, 16), (7, 17), (7, 18), (7, 19), (7, 20), (7, 21), (6, 21), (5, 21), (4, 21), (3, 21), (2, 21), (1, 21), (1, 20), (1, 19), (0, 19), (0, 20), (0, 21), (0, 22), (0, 23), (0, 24)  Cost: 712  Nodes: 103 |

1. **A\* Search**

A\* search was implemented using a priority queue, where the data type was a pair, with the first element being an integer to store the F cost, and the second element being a pair to store the corresponding (x, y) coordinates. The F cost was calculated as the summation of the G cost and H cost, where G cost is the actual cost from the start node to the current node, while the H cost is the estimated cost from the current node to the goal node. In particular, the heuristic function (H cost) was determined using the Manhattan distance, namely the sum of the absolute values of the differences between the current node and goal node. The priority queue was modified to work as a min heap, meaning the data is sorted by lowest F cost, with the top element having the lowest one. When data is popped from the min heap, the next lowest F cost value is moved to the top, and when data is added to the heap, it automatically inserts it into the correct position, again based on the F cost.

|  |
| --- |
| **Starting at S and ending at E1** |
| Path: (13, 2), (13, 3), (12, 3), (11, 3), (10, 3), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (10, 7), (11, 7), (11, 8), (11, 9), (12, 9), (13, 9), (13, 10), (13, 11), (12, 11), (11, 11), (11, 12), (11, 13), (10, 13), (9, 13), (8, 13), (8, 12), (8, 11), (7, 11), (6, 11), (5, 11), (5, 12), (5, 13), (5, 14), (5, 15), (5, 16), (5, 17), (5, 18), (5, 19), (5, 20), (5, 21), (5, 22), (5, 23),  Cost: 70  Nodes: 360 |

|  |
| --- |
| **Starting at S and ending at E2** |
| Path: (13, 2), (13, 3), (12, 3), (11, 3), (10, 3), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (8, 7), (7, 7), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (6, 2), (5, 2), (4, 2), (3, 2),  Cost: 49  Nodes: 278 |

|  |
| --- |
| **Starting at (0, 0) and ending at (24, 24)** |
| Path: (24, 0), (23, 0), (22, 0), (21, 0), (21, 1), (21, 2), (21, 3), (20, 3), (19, 3), (18, 3), (18, 4), (18, 5), (17, 5), (16, 5), (15, 5), (14, 5), (13, 5), (12, 5), (11, 5), (10, 5), (9, 5), (9, 6), (9, 7), (10, 7), (11, 7), (11, 8), (11, 9), (12, 9), (13, 9), (13, 10), (13, 11), (12, 11), (11, 11), (11, 12), (11, 13), (10, 13), (9, 13), (8, 13), (8, 12), (8, 11), (7, 11), (6, 11), (5, 11), (4, 11), (3, 11), (2, 11), (1, 11), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (1, 16), (1, 17), (1, 18), (1, 19), (0, 19), (0, 20), (0, 21), (0, 22), (0, 23), (0, 24),  Cost: 91  Nodes: 381 |

# Question 3

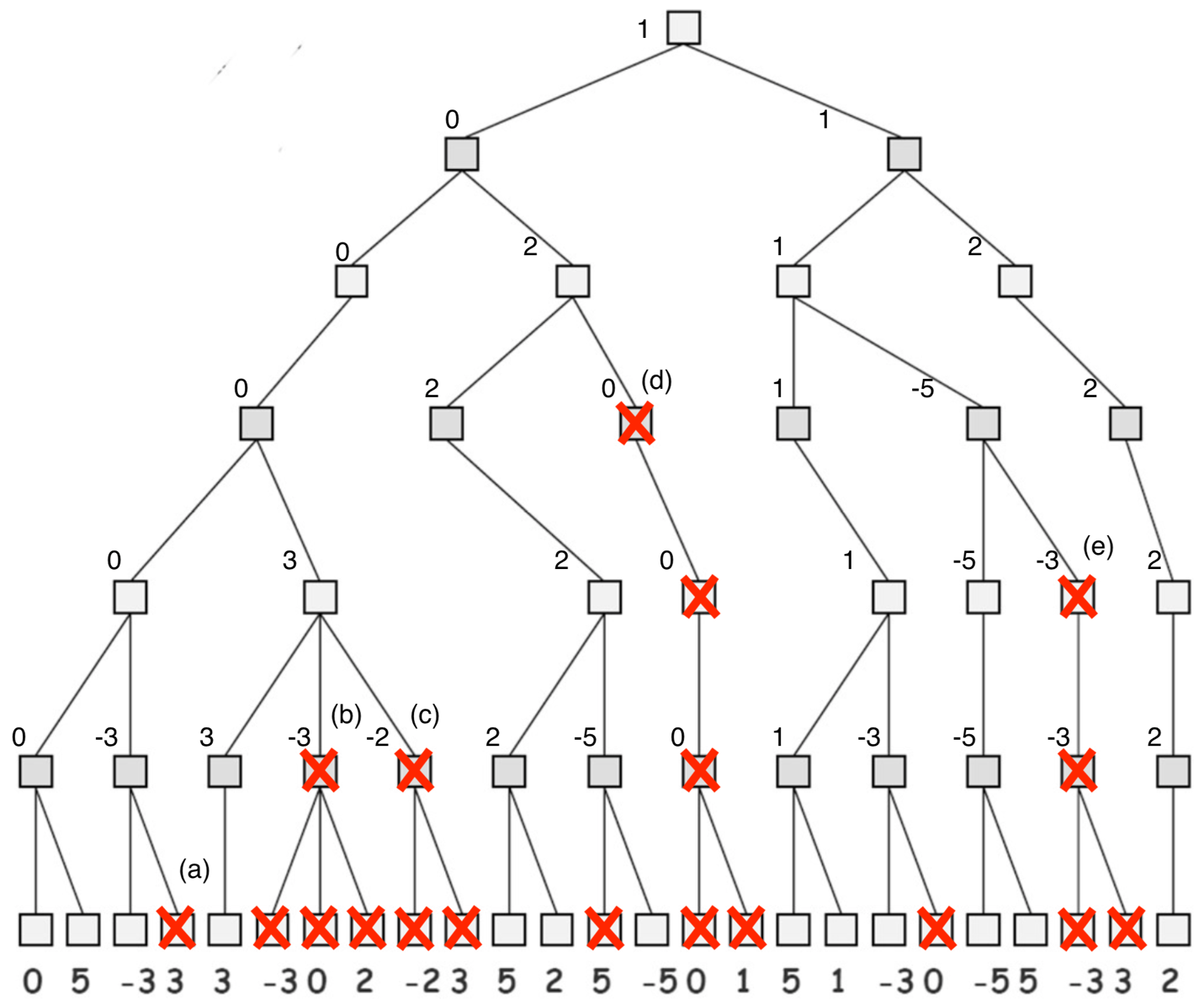
1. **Backed-Up Values by Minimax Algorithm**

A picture containing diagram

Description automatically generated

Since the root node corresponds to the maximizing player, then all light-coloured nodes represent max levels, whereas all dark-coloured nodes represent min levels. The strategy is to back up the highest value nodes for maximum levels and lowest value nodes for minimum levels.

1. **Pruning by Alpha-Beta Pruning Algorithm**

****

Since and on max level, can prune 3 since will take 0 no matter the value

Since and on min level, can prune -3 and its children since will always be 0

Since and on min level, can prune -2 and its children since will always be 0

Since and on min level, can prune 0 and its children since will always be 0

Since and on max level, can prune -3 and its children since will always be 1

1. **Tree Restructuring to Increase Pruned Branches**

**A picture containing light, train, street, traffic

Description automatically generated**

The tree has been restructured by connecting the branch with the 2’s, far on the right, at the second min level, to the -5 node. This increases the number of pruned branches from 19 to 23. In this case, at the second max level and at the second min level, so this newly added 2 node, as well as all its associated children, can be pruned. This is because it does not matter what its value is, as will always be taking the 1.

# Question 4