

CSE_4082 Project 1 Design Document

Alperen Bayraktar 150116501

Onur Can Yücedağ 150116825

	1	2	3	4	5	6	7	8
1								
2				T				
3		S				T	G	
4								
5								
6							G	
7						T		T
8					G			G

Creation of the Map:

```
int idx = 0;
std::string word;

StateMatrix.clear();
StateMatrix.resize(WIDTH, std::vector<Node*>(HEIGHT));
int start_row = NULL, start_column = NULL;

// Read every word in input file.
while (inFile >> word)
{
    // Assign current row and column.
    int row = idx / HEIGHT;
    int column = idx % WIDTH;

    // Parse node features and create a new node.
    auto new_node_features = ParseNodeFeatures(word, row, column);
    Node* new_node = new Node(new_node_features);

    // Populate StateMatrix.
    StateMatrix[row][column] = new_node;
    idx++;

    // Get start column and row and populate GoalNodes.
    if (new_node_features->type == 'S') {
        start_column = column;
        start_row = row;
    }
    else if (new_node_features->type == 'G') {
        GoalNodes.emplace_back(new_node);
    }
}
```

In main(), the .txt file we've given as input gets transformed into a 8x8 matrix called "StateMatrix".

inFile is the maze.txt that represents the maze cells in words.

To do the transformation, "ParseNodeFeatures" method is used. A node is getting created with the features(struct) as a return value.

Then this new node is added to the StateMatrix.

Index gets incremented and then the type control is made.

If the type of a cell is "S", then that index of the StateMatrix is the start point.

If the type of a cell is "G", then that index of the StateMatrix is a goal point.

```
// Parses each word from the input file to form a NodeFeatures struct.
struct NodeFeatures* ParseNodeFeatures(std::string word, int x, int y) {
    int cost = NULL;
    char type = NULL;

    if (word[0] == '1' || word[0] == '7') {
        cost = word[0] - '0';
        type = word[0] == '1' ? 'N' : 'T';
    }
    else if (word[0] == 'S') {
        cost = 0;
        type = word[0];
    }
    else {
        cost = 1;
        type = word[0];
    }

    bool explored = false;
    bool frontiered = false;
    short west = (short)(word[1] == '.');
    short north = (short)(word[2] == '.');
    short east = (short)(word[3] == '.');
    short south = (short)(word[4] == '.');
    int depth = 0;
    struct NodeFeatures* node_feats = new NodeFeatures{ cost, x, y, type, west, north, east, south, explored, frontiered, depth };
    return node_feats;
}
```

maze.txt - Not Defteri

Dosya Düzen Biçim Görünüm Yardım

```
1;. 1;. 1;. 1;. 1;. 1;. 1;. 1;.
1;. 1;. 1;. 7;. 1;. 1;. 1;. 1;.
1;. S;. 1;. 1;. 1;. 7;. 6;. 1;.
1;. 1;. 1;. 1;. 1;. 1;. 1;. 1;.
1;. 1;. 1;. 1;. 1;. 1;. 1;. 1;.
1;. 1;. 1;. 1;. 1;. 6;. 1;. 1;.
1;. 1;. 1;. 1;. 7;. 1;. 7;. 1;.
1;. 1;. 1;. 1;. 6;. 1;. 6;. 1;.
```

First letter represents the type, thus the cost.

The rest 4 letters are the direction status in respect to “west, north, east, south” . “,” if it’s blocked and “.” If it’s not blocked.

Costs and types are set according to the first letter than directions are converted to 1’s and 0’s.

The explored and frontiered booleans are set to false for initialization.

Finally, the struct gets created and fed to the return.

Node Struct and class in the header file:

```
12 struct NodeFeatures {
13     int cost;
14     int x, y;
15     char type;
16     short west, north, east, south;
17     bool explored, frontiered;
18     int depth;
19 };
20 enum SearchAlgorithm { BFS, DFS, IDS, UCS, ASTAR, GBFS, DLS };
21 class Node {
22 public:
23     int cost;
24     int x, y;
25     Node* parent;
26     char type;
27     short west, north, east, south;
28     bool explored;
29     bool frontiered;
30     int depth;
31     Node(struct NodeFeatures* nodefeats)
32     : x(nodefeats->x),
33       y(nodefeats->y),
34       west(nodefeats->west),
35       north(nodefeats->north),
36       east(nodefeats->east),
37       south(nodefeats->south),
38       cost(nodefeats->cost),
39       type(nodefeats->type),
40       explored(nodefeats->explored),
41       frontiered(nodefeats->frontiered),
42       depth(nodefeats->depth)
43     {}
44     ~Node() {}
45 };
```

```

581         if (StateMatrix[start_row][start_column]->type == 'G') {
582             ReturnPath(StateMatrix[start_row][start_column]);
583             return 1;
584         }
585         std::vector<Node*> frontier;
586         std::vector<Node*> explored;
587
588         StateMatrix[start_row][start_column]->frontiered = true;
589         frontier.emplace_back((StateMatrix[start_row][start_column]));
590

```

In main(), as a start, if the start node is also Goal, the program exits with success.

Frontier and expanded lists are created.

Start node gets added to frontier list.

Then, according to the user input, a method is implemented on the maze:

```

while (true)
{
    std::cout << "\n >>>>>>>>>>";
    std::cout << "\n Menu";
    std::cout << "\n =====";
    std::cout << "\n BFS - 1";
    std::cout << "\n UCS - 2";
    std::cout << "\n DFS - 3";
    std::cout << "\n IDS - 4";
    std::cout << "\n GBFS - 5";
    std::cout << "\n ASTAR - 6";
    std::cout << "\n Exit - 0";
    std::cout << "\n Enter selection: ";

```

This is a menu-driven code.

<pre> 591 switch (selection) { 592 case 1: 593 { 594 Node* result = ExecuteBFS(StateMatrix, frontier, explored, true); 595 if (result) { 596 ReturnPath(result); 597 } 598 } 599 break; 600 case 2: 601 { 602 Node* result = ExecuteUCS(StateMatrix, frontier, explored, true); 603 if (result) { 604 ReturnPath(result); 605 } 606 } 607 break; 608 case 3: 609 { 610 Node* result = ExecuteDFS(StateMatrix, frontier, explored, true); 611 if (result) { 612 ReturnPath(result); 613 } 614 } 615 break; 616 case 4: 617 { 618 Node* result = ExecuteIDS(StateMatrix, frontier, explored, true); 619 if (result) { 620 ReturnPath(result); 621 } 622 } 623 break; 624 case 5: </pre>	<pre> 624 case 5: 625 { 626 Node* result = ExecuteGBFS(StateMatrix, frontier, explored, true); 627 if (result) { 628 ReturnPath(result); 629 } 630 } 631 break; 632 case 6: 633 { 634 Node* result = ExecuteASTAR(StateMatrix, frontier, explored, true); 635 if (result) { 636 ReturnPath(result); 637 } 638 } 639 break; 640 case 0: 641 return 0; 642 break; 643 default: 644 { 645 std::cout << "\n !!! Invalid selection \n"; 646 } 647 } 648 GlobalExplored.clear(); 649 } 650 return 0; 651 } 652 </pre>
---	---

Line 648 is for IDS.

```

224 // Left shift overloading for displaying a node with standard library.
225 std::ostream& operator << (std::ostream& out, Node* c)
226 {
227     out << "| cost: " << c->cost;
228     out << "| x, y: " << "(" << c->x + 1 << ", " << c->y + 1 << ")";
229     out << "| type: " << c->type;
230     out << "| explored: " << c->explored;
231     out << "| frontier: " << c->frontier;
232     return out;
233 }
234
235 // Returns the path that is starting from leaf/child node to its greatest parent and displays the information of each travelled node.
236 // Also displays the total cost of the path.
237 void ReturnPath(Node* end_node) {
238     int total_cost = 0;
239     Node* current_printed_node;
240     current_printed_node = end_node;
241     std::cout << "Path Found" << std::endl;
242     while (current_printed_node != NULL) {
243         if (current_printed_node->type != 'S')
244             total_cost += current_printed_node->cost;
245         std::cout << current_printed_node << std::endl;
246         current_printed_node = current_printed_node->parent;
247     }
248     std::cout << "Total Cost: " << total_cost << std::endl;
249 }
250

```

ReturnPath() function is for printing both the path to goal state and the cost. We've overloaded the print of a node and used it in the method. It simply goes to parent until it is the start cell.

```

252 // Visualizes/Displays the vector of Node* kind.
253 void VisualizeVector(std::vector<Node*>& frontier) {
254     std::cout << "-----\n";
255     for (auto cur_frontier : frontier) {
256         std::cout << "(" << cur_frontier->x + 1 << ", " << cur_frontier->y + 1 << ")\n";
257     }
258     std::cout << "-----\n";
259 }
260
261

```

This method is for printing the vectors like expanded or frontier.

```

51 Node* ActionSpace(Node* current_node, std::vector<Node*> &frontie
52 {
53     std::vector<Node*> action_vector;
54     int x = current_node->x;
55     int y = current_node->y;
56     if (current_node->east) {
57         action_vector.emplace_back(StateMatrix[x][y + 1]);
58     }
59     if (current_node->south) {
60
61         action_vector.emplace_back(StateMatrix[x + 1][y]);
62     }
63     if (current_node->west) {
64
65         action_vector.emplace_back(StateMatrix[x][y - 1]);
66     }
67     if (current_node->north) {
68
69         action_vector.emplace_back(StateMatrix[x - 1][y]);
70     }
71     switch (search_algorithm)
72     {

```

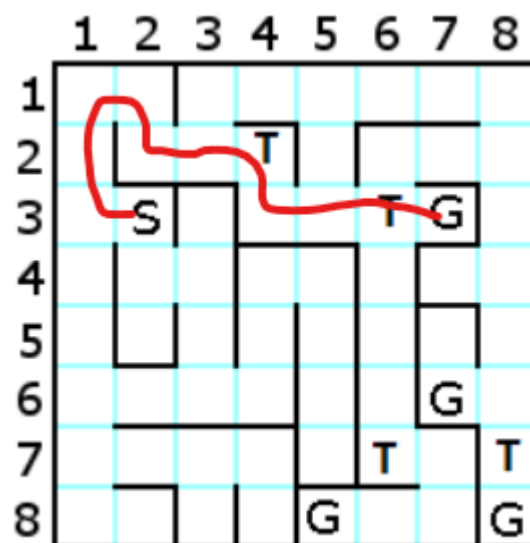
ActionSpace() method is for creating the frontier list for each algorithm, it has modifications for each algorithm.

Since the given priority is east->south->west->north, it adds nodes to the action vector respect to this.

The cases will be explained in each algorithms own section...

- Frontier list for each iteration is printed.
- Expanded list is printed.
- Total cost is printed.

1- Depth First Search



And the path it followed:

```
Path Found
| cost: 1 | x, y: (3, 7) | type: G | explored: 1 | frontiered: 0
| cost: 7 | x, y: (3, 6) | type: T | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 5) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 4) | type: N | explored: 1 | frontiered: 0
| cost: 7 | x, y: (2, 4) | type: T | explored: 1 | frontiered: 0
| cost: 1 | x, y: (2, 3) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (2, 2) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (1, 2) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (1, 1) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (2, 1) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 1) | type: N | explored: 1 | frontiered: 0
| cost: 0 | x, y: (3, 2) | type: S | explored: 1 | frontiered: 0
Total Cost: 23
```

Explored Set:

```
-----
(3, 2)
(4, 2)
(4, 3)
(5, 3)
(6, 3)
(6, 4)
(5, 4)
(4, 4)
(4, 5)
(5, 5)
(6, 5)
(7, 5)
(6, 2)
(6, 1)
(7, 1)
(7, 2)
(7, 3)
(7, 4)
(8, 4)
(8, 3)
(8, 1)
(8, 2)
(5, 1)
(4, 1)
(3, 3)
(5, 2)
(3, 1)
(2, 1)
(1, 1)
(1, 2)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(3, 5)
(3, 6)
```

(Expanded Nodes)

```
-----
(3, 1)
(4, 2)
-----
(3, 1)
(5, 2)
(4, 3)
-----
(3, 1)
(5, 2)
(3, 3)
(5, 3)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 3)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 2)
(6, 4)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 2)
(5, 4)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 2)
(4, 4)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 2)
(4, 5)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 2)
(5, 5)
-----
(3, 1)
(5, 2)
(3, 3)
(6, 2)
(6, 5)
```

(Frontier list per iteration)

```

264 Node* ExecuteDFS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
265 {
266     int i = 0;
267     std::cout << "DFS is selected." << std::endl;
268     while (true)
269     {
270         // If frontier is empty, there is something wrong.
271         if (frontier.empty()) {
272             std::cout << "Failure... Path didn't found.\n";
273             return NULL;
274         }
275         // Get the current node from the last element of frontier.
276         Node* current_node;
277         current_node = frontier[frontier.size() - 1];
278         current_node->frontiered = false;
279
280         // Erase the last element, pop it.
281         frontier.erase(frontier.end() - 1);
282
283         // Add the current node to the explored set.
284         explored.emplace_back(current_node);
285         current_node->explored = true;
286
287         // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
288         Node* result_child = ActionSpace(current_node, frontier, StateMatrix, DFS, visualize_vector);
289         if (result_child) {
290             std::cout << "\nExplored Set:\n";
291             VisualizeVector(explored);
292             return result_child;
293         }
294     }
295 }
296

```

The last node gets extracted from the frontier.

Gets added to expanded list.

Then gets fed to ActionSpace() method to get its children to frontier.

```

91 case DFS:
92 {
93
94     std::reverse(action_vector.begin(), action_vector.end());
95     for (Node* cur_child : action_vector)
96     {
97         if (!cur_child->explored && !cur_child->frontiered) {
98             cur_child->parent = current_node;
99             if (cur_child->type == 'G') {
100                 cur_child->explored = true;
101                 return cur_child;
102             }
103             cur_child->frontiered = true;
104             frontier.emplace_back(cur_child);
105         }
106     }
107
108     if (visualize_vector)
109         VisualizeVector(frontier);
110 }
111 break;

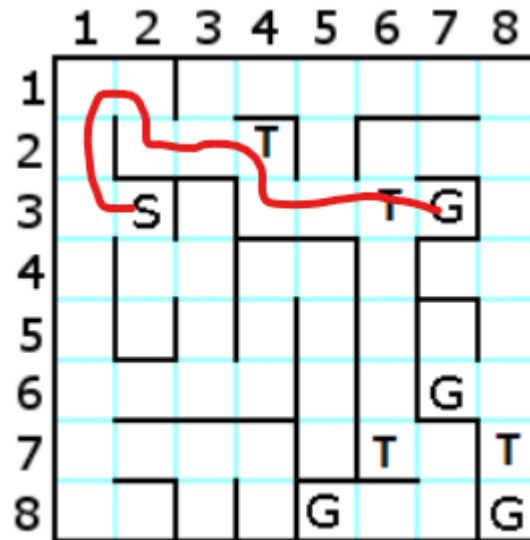
```

Action Vector has the current nodes children, so it needs to get reversed because DFS uses a stack data structure.

If the child is not in frontier or expanded list, it gets added to frontier list if it is not a Goal state.

Then frontier list gets printed by VisualizeVector().

2- Breadth First Search



Explored Set:

 (3, 2)
 (4, 2)
 (3, 1)
 (4, 3)
 (5, 2)
 (4, 1)
 (2, 1)
 (5, 3)
 (3, 3)
 (5, 1)
 (1, 1)
 (6, 3)
 (6, 1)
 (1, 2)
 (6, 4)
 (6, 2)
 (7, 1)
 (2, 2)
 (5, 4)
 (7, 2)
 (8, 1)
 (2, 3)
 (4, 4)
 (7, 3)
 (8, 2)
 (2, 4)
 (1, 3)
 (4, 5)
 (7, 4)
 (8, 3)
 (3, 4)
 (1, 4)
 (5, 5)
 (8, 4)
 (3, 5)
 (1, 5)
 (6, 5)
 (3, 6)

Path Found

cost: 1	x, y: (3, 7)	type: G	explored: 1	frontiered: 0
cost: 7	x, y: (3, 6)	type: T	explored: 1	frontiered: 0
cost: 1	x, y: (3, 5)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (3, 4)	type: N	explored: 1	frontiered: 0
cost: 7	x, y: (2, 4)	type: T	explored: 1	frontiered: 0
cost: 1	x, y: (2, 3)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 2)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 2)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 1)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 1)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (3, 1)	type: N	explored: 1	frontiered: 0
cost: 0	x, y: (3, 2)	type: S	explored: 1	frontiered: 0

Total Cost: 23


```

298 Node* ExecuteBFS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
299 {
300     int i = 0;
301     std::cout << "BFS is selected." << std::endl;
302     while (true)
303     {
304         // If frontier is empty, there is something wrong.
305         if (frontier.empty()) {
306             std::cout << "Failure... Path didn't found.\n";
307             return NULL;
308         }
309         // Get the current node from the first element of frontier.
310         Node* current_node;
311         current_node = frontier[0];
312         current_node->frontiered = false;
313         // Erase the first element, pop it.
314         frontier.erase(frontier.begin());
315
316         // Add the current node to the explored set.
317         explored.emplace_back(current_node);
318         current_node->explored = true;
319
320         // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
321         Node* result_child = ActionSpace(current_node, frontier, StateMatrix, BFS, visualize_vector);
322         if (result_child) {
323             std::cout << "\nExplored Set:\n";
324             VisualizeVector(explored);
325             return result_child;
326         }
327     }
328 }

```

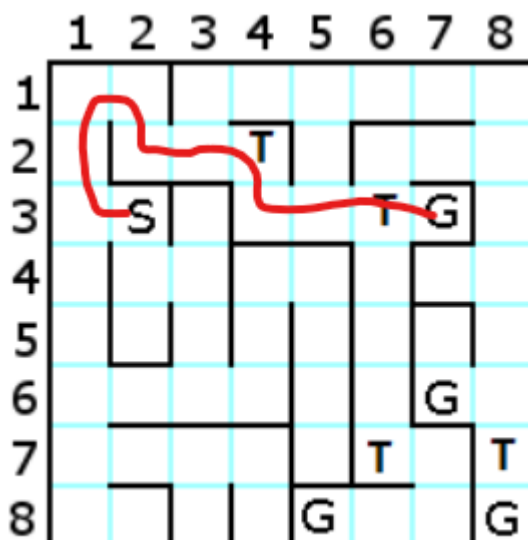
Every step is same with DFS except that the last node getting extracted. This time the first node gets extracted every time and ActionSpace() adds the nodes in their normal order.

```

73 case BFS:
74 {
75     for (Node* cur_child : action_vector)
76     {
77         if (!cur_child->explored && !cur_child->frontiered) {
78             cur_child->parent = current_node;
79             if (cur_child->type == 'G') {
80                 cur_child->explored = true;
81                 return cur_child;
82             }
83             cur_child->frontiered = true;
84             frontier.emplace_back(cur_child);
85         }
86     }
87     if (visualize_vector)
88         VisualizeVector(frontier);
89 }
90 break;

```

3- Iterative Deepening



```

Path Found
| cost: 1 | x, y: (3, 7) | type: G | explored: 1 | frontiered: 0
| cost: 7 | x, y: (3, 6) | type: T | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 5) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 4) | type: N | explored: 1 | frontiered: 0
| cost: 7 | x, y: (2, 4) | type: T | explored: 1 | frontiered: 0
| cost: 1 | x, y: (2, 3) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (2, 2) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (1, 2) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (1, 1) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (2, 1) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 1) | type: N | explored: 1 | frontiered: 0
| cost: 0 | x, y: (3, 2) | type: S | explored: 1 | frontiered: 0
Total Cost: 23

```

Explored set:

```

-----
(3, 2)
(4, 2)
(4, 3)
(5, 3)
(6, 3)
(6, 4)
(5, 4)
(4, 4)
(4, 5)
(5, 5)
(6, 5)
(7, 5)
(6, 2)
(6, 1)
(7, 1)
(7, 2)
(7, 3)
(7, 4)
(8, 4)
(8, 3)
(8, 1)
(8, 2)
(5, 1)
(4, 1)
(3, 3)
(5, 2)
(3, 1)
(2, 1)
(1, 1)
(1, 2)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(3, 5)
(3, 6)
(3, 7)
-----

```

To trace recursiveness, we've added some print outs.

```

Trying for limit: 0
We are at node: 3, 2
Limit reached.and the node was: 3, 2
Trying for limit: 1
We are at node: 3, 2
-----
(3, 1)
(4, 2)
-----
We are at node: 4, 2
Limit reached.and the node was: 4, 2
We are at node: 3, 1
Limit reached.and the node was: 3, 1
Returning back to upper depth.
Trying for limit: 2
We are at node: 3, 2
-----
(3, 1)
(4, 2)
-----
We are at node: 4, 2
-----
(5, 2)
(4, 3)
-----
We are at node: 4, 3
Limit reached.and the node was: 4, 3
We are at node: 5, 2
Limit reached.and the node was: 5, 2
Returning back to upper depth.
We are at node: 3, 1
-----
(2, 1)
(4, 1)
-----
We are at node: 4, 1
Limit reached.and the node was: 4, 1
We are at node: 2, 1
Limit reached.and the node was: 2, 1
Returning back to upper depth.
Returning back to upper depth.
Trying for limit: 3
We are at node: 3, 2
-----
(3, 1)
(4, 2)
-----
We are at node: 4, 2
-----
(5, 2)
(4, 3)
-----
We are at node: 4, 3
-----
(3, 3)
(5, 3)
-----

```

The implementation was made similar to the course slides.

```
448 Node* ExecuteIDS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
449 {
450     //Initialize the depth as 0.
451     int depth = 0;
452     std::cout << "IDS is selected." << std::endl;
453     while (true)
454     {
455         std::cout << "Trying for limit: " << depth << std::endl;
456         //Set result as the output of recursive function.
457         Node* result = ExecuteDLS(StateMatrix, frontier, explored, visualize_vector, depth);
458         depth++;
459         if (result) {
460             std::cout << "Explored set:" << std::endl;
461             //Prints the final true ordered expanded queue.
462             VisualizeVector(GlobalExplored);
463             return result;
464         }
465         //In the recursive part, it never sets the explored bool of nodes to false so we need to set it all to false here for further expansions.
466         for (int i = 0; i < StateMatrix.size(); i++)
467         {
468             for (int j = 0; j < StateMatrix[i].size(); j++)
469             {
470                 StateMatrix[i][j]->explored = false;
471             }
472         }
473         //If there is a limit increase again, it clears the current explored set.
474         GlobalExplored.clear();
475     }
476 }
```

In ExecuteIDS(), depth gets instantiated as 0.

Then in infinite loop, limit gets increased and recursive function gets executed.

If the goal state is reached, explored list gets printed out and function returns success.

If the goal state is not reached, then every nodes explored Boolean is set false, so program can work correctly. Also explored list is cleared.

```
510 // Execution of DLS for IDS.
511 Node* ExecuteDLS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector, int limit)
512 {
513     return RecursiveDLS(StateMatrix, frontier, explored, visualize_vector, limit);
514 }
```

```
470 Node* RecursiveDLS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector, int limit)
471 {
472     Node* result;
473     if (frontier.empty()) return NULL;
474     Node* current_node;
475     current_node = frontier[frontier.size() - 1];
476     current_node->frontiered = false;
477     frontier.erase(frontier.end() - 1);
478     std::cout << "We are at node: " << current_node->x + 1 << ", " << current_node->y + 1 << std::endl;
479     explored.emplace_back(current_node);
480     current_node->explored = true;
481     //When the node is extracted from the frontier list we add it to global expanded list.
482     GlobalExplored.emplace_back(current_node);
483     if (current_node->type == 'G') return current_node;
484     else if (limit == 0) {
485         //If the function is on the leaf node for the given depth limit, it returns null.
486         std::cout << "Limit reached." << "and the node was: " << current_node->x + 1 << ", " << current_node->y + 1 << std::endl;
487         return NULL;
488     }
489     else {
490         //Gets the child nodes of the current node.
491         Node* result_child = ActionSpace(current_node, frontier, StateMatrix, DLS, visualize_vector);
492         for (Node* cur_child : frontier)
493         {
494             //for each node in the action space list, it executes the recursive function again.
495             std::vector<Node*> frontierRec;
496             std::vector<Node*> exploredRec;
497             frontierRec.emplace_back(frontier[frontier.size() - 1]);
498             //Need to erase the last node from the list, so it can check the node before it at the next iteration.
499             frontier.erase(frontier.end() - 1);
500             result = RecursiveDLS(StateMatrix, frontierRec, exploredRec, visualize_vector, limit - 1);
501             //If a goal node is reached, returns the node.
502             if (result) return result;
503         }
504         //If the current frontier list of the node is empty, it means that we need to go up in the tree.
505         if (frontier.size() == 0) std::cout << "Returning back to upper depth." << std::endl;
506         return NULL;
507     }
508 }
```

If frontier is empty, it means either failure or program needs to go to upper depths.

Node gets extracted from frontier list and gets added to explored.

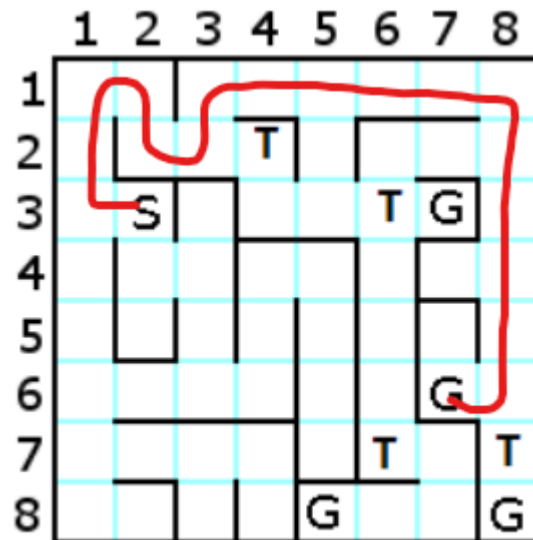
If the nodes limit is 0, the program can't go to lower depths and cuts off.

If the node still has limit, it gets fed to ActionSpace() function and gets its frontier list. This time, goal state check is made in this function only.

New expanded and frontier lists are created for the current node.

Then for each child, node gets added to new frontier list (acts like a start node itself)
 When the original frontier list is empty this loop ends, if a goal state is reached, it returns the node.
 If frontier size is 0, then program goes back to the upper depth.

4- Uniform Cost Search



Explored Set:

```

-----
(3, 2)
(4, 2)
(3, 1)
(4, 3)
(5, 2)
(4, 1)
(2, 1)
(5, 3)
(3, 3)
(5, 1)
(1, 1)
(6, 3)
(6, 1)
(1, 2)
(6, 4)
(6, 2)
(7, 1)
(2, 2)
(5, 4)
(7, 2)
(8, 1)
(2, 3)
(4, 4)
(7, 3)
(8, 2)
(1, 3)
(4, 5)
(7, 4)
(8, 3)
(1, 4)
(5, 5)
(8, 4)
(1, 5)
(6, 5)
(1, 6)
(2, 5)
(7, 5)
(1, 7)
(3, 5)
(1, 8)
(3, 4)
(2, 4)
(2, 8)
(3, 8)
(2, 7)
(4, 8)
(2, 6)
(5, 8)
(4, 7)
(6, 8)

```

Path Found

cost: 1	x, y: (6, 7)	type: G	explored: 1	frontiered: 0
cost: 1	x, y: (6, 8)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (5, 8)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (4, 8)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (3, 8)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 8)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 8)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 7)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 6)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 5)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 4)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 3)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 3)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 2)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 2)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 1)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 1)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (3, 1)	type: N	explored: 1	frontiered: 0
cost: 0	x, y: (3, 2)	type: S	explored: 1	frontiered: 0

Total Cost: 18

```

342 Node* ExecuteUCS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
343 {
344     std::cout << "UCS is selected." << std::endl;
345     while (true)
346     {
347         // If frontier is empty, there is something wrong.
348         if (frontier.empty()) {
349             std::cout << "Failure... Path didn't found.\n";
350             return NULL;
351         }
352     }
353
354     // Get the current node from the first element of frontier.
355     Node* current_node;
356     current_node = frontier[0];
357     current_node->frontiered = false;
358     // Erase the first element, pop it.
359     frontier.erase(frontier.begin());
360
361     // Add the current node to the explored set.
362     explored.emplace_back(current_node);
363     current_node->explored = true;
364
365     // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
366     Node* result_child = ActionSpace(current_node, frontier, StateMatrix, UCS, visualize_vector);
367     if (result_child) {
368         std::cout << "\nExplored Set:\n";
369         VisualizeVector(explored);
370         return result_child;
371     }
372 }
373

```

UCS works like BFS except the frontier gets sorted each iteration in ActionSpace().

```

184 case UCS:
185 {
186     for (Node* cur_child : action_vector)
187     {
188         if (!cur_child->explored && !cur_child->frontiered) {
189             cur_child->parent = current_node;
190             if (cur_child->type == 'G') {
191                 cur_child->explored = true;
192                 return cur_child;
193             }
194             cur_child->frontiered = true;
195             frontier.emplace_back(cur_child);
196         }
197     }
198     std::sort(frontier.begin(), frontier.end(), CompareTwoNodesCosts);
199     if (visualize_vector)
200         VisualizeVector(frontier);
201 }
202 break;

```

After, addition of new nodes, list gets sorted.

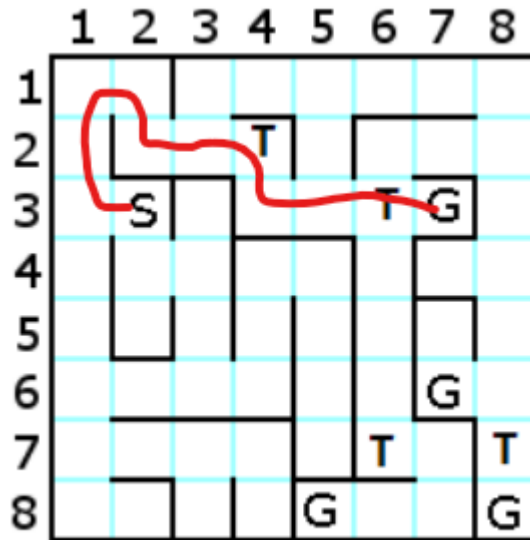
```

209 // Compares two nodes with their current path costs.
210 bool CompareTwoNodesCosts(Node* n1, Node* n2) {
211     return CurrentPathCost(n1) < CurrentPathCost(n2);
212 }

```

This comparison method is simply binary sort.

5- Greedy Best First Search



Path Found

cost: 1	x, y: (3, 7)	type: G	explored: 1	frontiered: 0
cost: 7	x, y: (3, 6)	type: T	explored: 1	frontiered: 0
cost: 1	x, y: (3, 5)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (3, 4)	type: N	explored: 1	frontiered: 0
cost: 7	x, y: (2, 4)	type: T	explored: 1	frontiered: 0
cost: 1	x, y: (2, 3)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 2)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 2)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (1, 1)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (2, 1)	type: N	explored: 1	frontiered: 0
cost: 1	x, y: (3, 1)	type: N	explored: 1	frontiered: 0
cost: 0	x, y: (3, 2)	type: S	explored: 1	frontiered: 0

Total Cost: 23

Explored Set:

```

-----
(3, 2)
(4, 2)
(4, 3)
(3, 3)
(5, 3)
(6, 3)
(6, 4)
(5, 4)
(4, 4)
(4, 5)
(5, 5)
(6, 5)
(7, 5)
(6, 2)
(3, 1)
(5, 2)
(6, 1)
(7, 1)
(7, 2)
(7, 3)
(7, 4)
(8, 4)
(8, 3)
(8, 1)
(8, 2)
(4, 1)
(2, 1)
(5, 1)
(1, 1)
(1, 2)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(3, 5)
(3, 6)
-----

```

```

374 // Execution of GBFS
375 Node* ExecuteGBFS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector) {
376
377     std::cout << "GBFS is selected." << std::endl;
378     while (true)
379     {
380         // If frontier is empty, there is something wrong.
381         if (frontier.empty()) {
382             std::cout << "Failure... Path didn't found.\n";
383             return NULL;
384         }
385         // Get the current node from the first element of frontier.
386         Node* current_node;
387         current_node = frontier[0];
388         current_node->frontiered = false;
389
390         // Erase the first element, pop it.
391         frontier.erase(frontier.begin());
392
393         // Add the current node to the explored set.
394         explored.emplace_back(current_node);
395         current_node->explored = true;
396
397         // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
398         Node* result_child = ActionSpace(current_node, frontier, StateMatrix, GBFS, visualize_vector);
399         std::cout << "\nExplored Set:\n";
400         VisualizeVector(explored);
401         if (result_child) {
402             return result_child;
403         }
404     }
405 }

```

ExecuteGBFS() does the same thing with BFS in its main function.

```

150
151 case GBFS:
152 {
153     for (Node* cur_child : action_vector)
154     {
155         if (!cur_child->explored && !cur_child->frontiered) {
156             cur_child->parent = current_node;
157             if (cur_child->type == 'G') {
158                 cur_child->explored = true;
159                 return cur_child;
160             }
161             cur_child->frontiered = true;
162             frontier.emplace_back(cur_child);
163         }
164     }
165     std::sort(frontier.begin(), frontier.end(), CompareTwoNodesHeuristics);
166     if (visualize_vector)
167         VisualizeVector(frontier);
168 }
169 break;

```

In ActionSpace(), it sorts the list according to their heuristic functions result.

```

214 // Compares two nodes with their heuristic path costs.
215 bool CompareTwoNodesHeuristics(Node* n1, Node* n2) {
216     return CalculateHeuristic(n1) < CalculateHeuristic(n2);
217 }

```

This function simply returns the result of Manhattan distance function comparison.

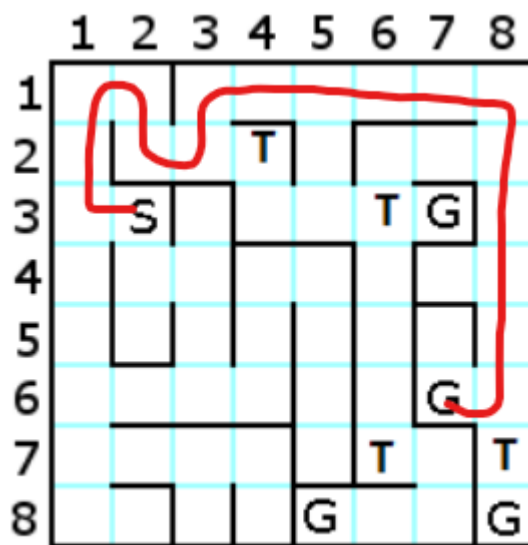

```

3 // Calculating the Manhattan Distance Heuristic.
4 int CalculateHeuristic(Node* current_node) {
5
6     if (current_node->type == 'G')
7         return 0;
8
9     int min_manhattan_distance = 9999999;
10
11     for (std::size_t i = 0; i < GoalNodes.size(); i++)
12     {
13         int cur_distance = abs(GoalNodes[i]->x - current_node->x) + abs(GoalNodes[i]->y - current_node->y);
14         if (cur_distance < min_manhattan_distance) {
15             min_manhattan_distance = cur_distance;
16         }
17     }
18     return min_manhattan_distance;
19 }
20

```

This function looks for the lowest Manhattan distance to the current cell the node's in.

6- A* Heuristic Search



Explored Set:

```

(3, 2)
(4, 2)
(3, 1)
(4, 3)
(3, 3)
(5, 2)
(5, 3)
(6, 3)
(6, 4)
(4, 1)
(2, 1)
(6, 2)
(5, 4)
(5, 1)
(1, 1)
(4, 4)
(1, 2)
(4, 5)
(2, 2)
(2, 3)
(6, 1)
(5, 5)
(7, 1)
(6, 5)
(7, 2)
(8, 1)
(7, 5)
(7, 3)
(8, 2)
(7, 4)
(8, 3)
(8, 4)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 5)
(1, 7)
(3, 5)
(1, 8)
(3, 4)
(2, 8)
(3, 8)
(2, 7)
(2, 4)
(4, 8)
(2, 6)
(4, 7)
(5, 8)
(6, 8)

```

Path Found

cost	x, y	type	explored	frontier
1	(6, 7)	G	1	0
1	(6, 8)	N	1	0
1	(5, 8)	N	1	0
1	(4, 8)	N	1	0
1	(3, 8)	N	1	0
1	(2, 8)	N	1	0
1	(1, 8)	N	1	0
1	(1, 7)	N	1	0
1	(1, 6)	N	1	0
1	(1, 5)	N	1	0
1	(1, 4)	N	1	0
1	(1, 3)	N	1	0
1	(2, 3)	N	1	0
1	(2, 2)	N	1	0
1	(1, 2)	N	1	0
1	(1, 1)	N	1	0
1	(2, 1)	N	1	0
1	(3, 1)	N	1	0
0	(3, 2)	S	1	0

Total Cost: 18


```

407 Node* ExecuteASTAR(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
408
409     std::cout << "ASTAR is selected." << std::endl;
410     while (true)
411     {
412         // If frontier is empty, there is something wrong.
413         if (frontier.empty()) {
414             std::cout << "Failure... Path didn't found.\n";
415             return NULL;
416         }
417         // Get the current node from the first element of frontier.
418         Node* current_node;
419         current_node = frontier[0];
420         current_node->frontiered = false;
421
422         // Erase the first element, pop it.
423         frontier.erase(frontier.begin());
424
425         // Add the current node to the explored set.
426         explored.emplace_back(current_node);
427         current_node->explored = true;
428
429         // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
430         Node* result_child = ActionSpace(current_node, frontier, StateMatrix, ASTAR, visualize_vector);
431         std::cout << "\nExplored Set:\n";
432         VisualizeVector(explored);
433         if (result_child) {
434             return result_child;
435         }
436     }
437 }

```

Same with GBFS.

```

131     case ASTAR:
132     {
133         for (Node* cur_child : action_vector)
134         {
135             if (!cur_child->explored && !cur_child->frontiered) {
136                 cur_child->parent = current_node;
137                 if (cur_child->type == 'G') {
138                     cur_child->explored = true;
139                     return cur_child;
140                 }
141                 cur_child->frontiered = true;
142                 frontier.emplace_back(cur_child);
143             }
144         }
145         std::sort(frontier.begin(), frontier.end(), CompareTwoNodesTotalCosts);
146         if (visualize_vector)
147             VisualizeVector(frontier);
148     }
149     break;

```

In ActionSpace(), after the new children are added to the frontier, the total cost is compared this time.

```

219 // Compares two nodes with their (heuristic + current) path costs.
220 bool CompareTwoNodesTotalCosts(Node* n1, Node* n2) {
221     return (CurrentPathCost(n1) + CalculateHeuristic(n1)) < (CurrentPathCost(n2) + CalculateHeuristic(n2));
222 }
223

```

This time, current cost is added to the heuristics.

```

331 int CurrentPathCost(Node* current_node) {
332
333     int total_cost = 0;
334     while (current_node)
335     {
336         total_cost += current_node->cost;
337         current_node = current_node->parent;
338     }
339     return total_cost;
340 }

```

Current cost is the cost of the path taken so far.