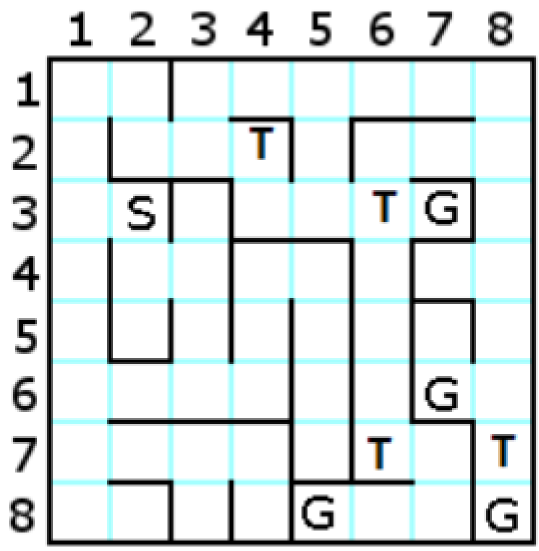
CSE_4082 Project 1 Design Document

Alperen Bayraktar 150116501

Onur Can Yücedağ 150116825



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.

```cpp
// 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....  G.;;;  1;.;.
1;.;.  1;...  1..;.  1;;..  1.;;.  1;.;.  1;;.;  1..;.
1;.;.  1;.;;  1;.;.  1;.;.  1;.;.  1;.;.  1;;;.  1;.;.
1;...  1.;.;  1...;  1..;;  1;.;.  1;.;.  G;..;  1..;.
1;...  1.;.;  1.;..  1.;;.  1;.;;  7;..;  1.;;.  7;.;.
1;..;  1.;;;  1;.;;  1;.;;  G;;.;  1.;.;  1..;;  G;.;;
```

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 blooleans are set to false for initialization.

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

Node Struct and class in the header file:

```cpp
struct NodeFeatures {
    int cost;
    int x, y;
    char type;
    short west, north, east, south;
    bool explored, frontiered;
    int depth;
};
enum SearchAlgorithm { BFS, DFS, IDS, UCS, ASTAR, GBFS, DLS };
class Node {
public:
    int cost;
    int x, y;
    Node* parent;
    char type;
    short west, north, east, south;
    bool explored;
    bool frontiered;
    int depth;
    Node(struct NodeFeatures* nodefeats)
        : x(nodefeats->x),
        y(nodefeats->y),
        west(nodefeats->west),
        north(nodefeats->north),
        east(nodefeats->east),
        south(nodefeats->south),
        cost(nodefeats->cost),
        type(nodefeats->type),
        explored(nodefeats->explored),
        frontiered(frontiered),
        depth(depth)
    {}
    ~Node() {}
};
```

```cpp
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:

```cpp
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.

```cpp
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:
```

```cpp
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 
```

Line 648 is for IDS.

```cpp
// Left shift overloading for displaying a node with standard library.
std::ostream& operator << (std::ostream& out, Node* c)
{
    out << "| cost: " << c->cost;
    out << " | x, y: " << "(" << c->x + 1 << ", " << c->y + 1 << ")";
    out << " | type: " << c->type;
    out << " | explored: " << c->explored;
    out << " | frontiered: " << c->frontiered;
    return out;
}

// Returns the path that is starting from leaf/child node to its greates parent and displays the information of each travelled node.
// Also displays the total cost of the path.
void ReturnPath(Node* end_node) {
    int total_cost = 0;
    Node* current_printed_node;
    current_printed_node = end_node;
    std::cout << "Path Found" << std::endl;
    while (current_printed_node != NULL) {

        if (current_printed_node->type != 'S')
            total_cost += current_printed_node->cost;
        std::cout << current_printed_node << std::endl;
        current_printed_node = current_printed_node->parent;
    }
    std::cout << "Total Cost: " << total_cost << std::endl;
}
```

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.

```cpp
// Visualizes/Displays the vector of Node* kind.
void VisualizeVector(std::vector<Node*>& frontier) {

    std::cout << "----------\n";
    for (auto cur_frontier : frontier) {

        std::cout << "(" << cur_frontier->x + 1 << ", " << cur_frontier->y + 1 << ")\n";
    }
    std::cout << "-----------\n";
}
```

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

```
Node* ActionSpace(Node* current_node, std::vector<Node*> &frontie
{
    std::vector<Node*> action_vector;
    int x = current_node->x;
    int y = current_node->y;
    if (current_node->east) {
        action_vector.emplace_back(StateMatrix[x][y + 1]);
    }
    if (current_node->south) {

        action_vector.emplace_back(StateMatrix[x + 1][y]);
    }
    if (current_node->west) {

        action_vector.emplace_back(StateMatrix[x][y - 1]);
    }
    if (current_node->north) {

        action_vector.emplace_back(StateMatrix[x - 1][y]);
    }
    switch (search_algorithm)
    {
```

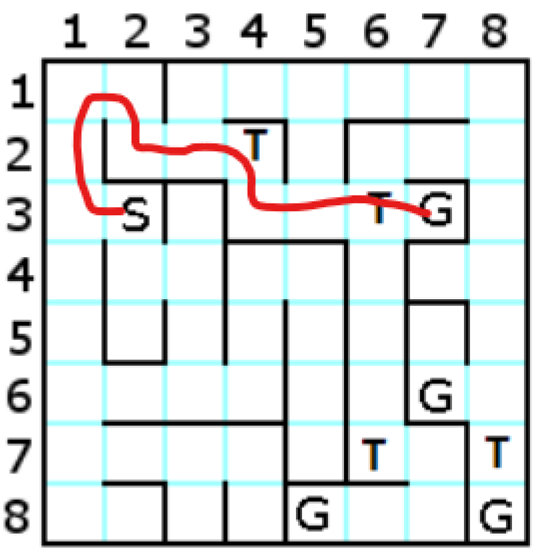
ActionSpace() method is for creating the frontier list for each algorithm, it has modifications for each algorithm.

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)

```cpp
Node* ExecuteDFS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
{
    int i = 0;
    std::cout << "DFS is selected." << std::endl;
    while (true)
    {
        // If frontier is empty, there is something wrong.
        if (frontier.empty()) {
            std::cout << "Failure... Path didn't found.\n";
            return NULL;
        }
        // Get the current node from the last element of frontier.
        Node* current_node;
        current_node = frontier[frontier.size() - 1];
        current_node->frontiered = false;

        // Erase the last element, pop it.
        frontier.erase(frontier.end() - 1);

        // Add the current node to the explored set.
        explored.emplace_back(current_node);
        current_node->explored = true;

        // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
        Node* result_child = ActionSpace(current_node, frontier, StateMatrix, DFS, visualize_vector);
        if (result_child) {
            std::cout << "\nExplored Set:\n";
            VisualizeVector(explored);
            return result_child;
        }
    }

}
```

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.

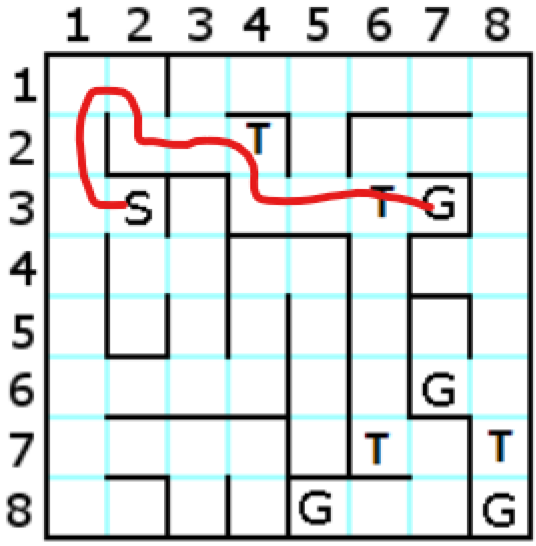
```cpp
        case DFS:
        {

            std::reverse(action_vector.begin(), action_vector.end());
            for (Node* cur_child : action_vector)
            {
                if (!cur_child->explored && !cur_child->frontiered) {
                    cur_child->parent = current_node;
                    if (cur_child->type == 'G') {
                        cur_child->explored = true;
                        return cur_child;
                    }
                    cur_child->frontiered = true;
                    frontier.emplace_back(cur_child);
                }
            }

            if (visualize_vector)
                VisualizeVector(frontier);
        }
        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)
```

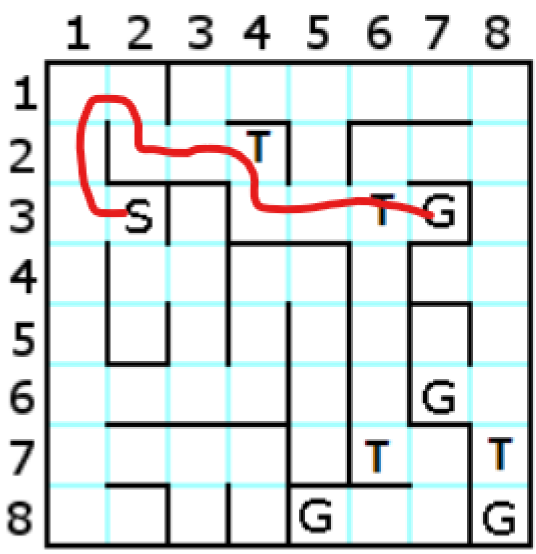
```
Path Found
| cost: 1 | x, y: (3, 7) | type: G | explored: 1 | frontiered: 0
| cost: 7 | x, y: (3, 6) | type: T | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 5) | type: N | explored: 1 | frontiered: 0
| cost: 1 | x, y: (3, 4) | type: N | explored: 1 | frontiered: 0
| cost: 7 | x, y: (2, 4) |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

```cpp
Node* ExecuteBFS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
{
    int i = 0;
    std::cout << "BFS is selected." << std::endl;
    while (true)
    {
        // If frontier is empty, there is something wrong.
        if (frontier.empty()) {
            std::cout << "Failure... Path didn't found.\n";
            return NULL;
        }
        // Get the current node from the first element of frontier.
        Node* current_node;
        current_node = frontier[0];
        current_node->frontiered = false;
        // Erase the first element, pop it.
        frontier.erase(frontier.begin());

        // Add the current node to the explored set.
        explored.emplace_back(current_node);
        current_node->explored = true;

        // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
        Node* result_child = ActionSpace(current_node, frontier, StateMatrix, BFS, visualize_vector);
        if (result_child) {
            std::cout << "\nExplored Set:\n";
            VisualizeVector(explored);
            return result_child;
        }
    }
}
```

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.

```cpp
    case BFS:
    {
        for (Node* cur_child : action_vector)
        {
            if (!cur_child->explored && !cur_child->frontiered) {
                cur_child->parent = current_node;
                if (cur_child->type == 'G') {
                    cur_child->explored = true;
                    return cur_child;
                }
                cur_child->frontiered = true;
                frontier.emplace_back(cur_child);
            }
        }
        if (visualize_vector)
            VisualizeVector(frontier);
    }
    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.

```cpp
Node* ExecuteIDS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
{
    //Initialize the depth as 0.
    int depth = 0;
    std::cout << "IDS is selected." << std::endl;
    while (true)
    {
        std::cout << "Trying for limit: " << depth << std::endl;
        //Set result as the output of recursive function.
        Node* result = ExecuteDLS(StateMatrix, frontier, explored, visualize_vector, depth);
        depth++;
        if (result) {
            std::cout << "Explored set:" << std::endl;
            //Prints the final true ordered expanded queue.
            VisualizeVector(GlobalExplored);
            return result;
        }
        //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.
        for (int i = 0; i < StateMatrix.size(); i++)
        {
            for (int j = 0; j < StateMatrix[i].size(); j++)
            {
                StateMatrix[i][j]->explored = false;
            }
        }
        //If there is a limit increase again, it clears the current explored set.
        GlobalExplored.clear();
    }
}
```

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.

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

```cpp
Node* RecursiveDLS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector, int limit)
{
    Node* result;
    if (frontier.empty()) return NULL;
    Node* current_node;
    current_node = frontier[frontier.size() - 1];
    current_node->frontiered = false;
    frontier.erase(frontier.end() - 1);
    std::cout << "We are at node: " << current_node->x + 1 << ", " << current_node->y + 1 << std::endl;
    explored.emplace_back(current_node);
    current_node->explored = true;
    //When the node is extracted from the frontier list we add it to global expanded list.
    GlobalExplored.emplace_back(current_node);
    if (current_node->type == 'G') return current_node;
    else if (limit == 0) {
        //If the function is on the leaf node for the given depthh limit, it returns null.
        std::cout << "Limit reached." << "and the node was: " << current_node->x + 1 << ", " << current_node->y + 1 << std::endl;
        return NULL;
    }
    else {
        //Gets the child nodes of the current node.
        Node* result_child = ActionSpace(current_node, frontier, StateMatrix, DLS, visualize_vector);
        for (Node* cur_child : frontier)
        {
            //for each node in the action space list, it executes the recursive function again.
            std::vector<Node*> frontierRec;
            std::vector<Node*> exploredRec;
            frontierRec.emplace_back(frontier[frontier.size() - 1]);
            //Need to erase the last node from the list, so it can check the node before it at the next iteration.
            frontier.erase(frontier.end() - 1);
            result = RecursiveDLS(StateMatrix, frontierRec, exploredRec, visualize_vector, limit - 1);
            //If a goal node is reached, returns the node.
            if (result) return result;
        }
        //If the currenct frontier list of the node is empty, it means that we need to go up in the tree.
        if (frontier.size() == 0) std::cout << "Returning back to upper depth." << std::endl;
        return NULL;
    }

}
```

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.

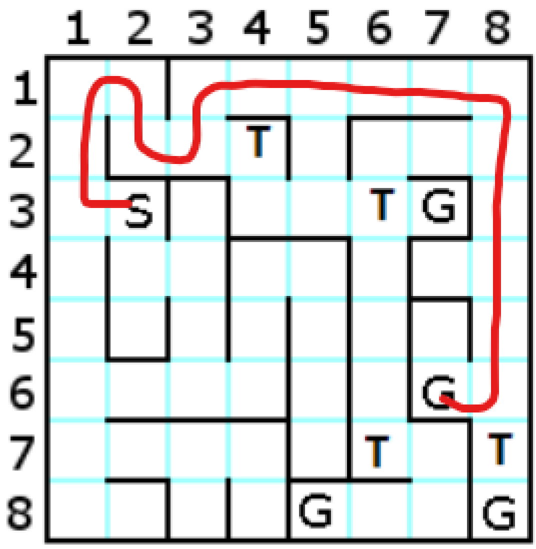
If the nodes limit is 0, the program can’t go to lower depths and cuts off.

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
```

```cpp
Node* ExecuteUCS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
{

    std::cout << "UCS is selected." << std::endl;
    while (true)
    {
        // If frontier is empty, there is something wrong.
        if (frontier.empty()) {
            std::cout << "Failure... Path didn't found.\n";
            return NULL;
        }

        // Get the current node from the first element of frontier.
        Node* current_node;
        current_node = frontier[0];
        current_node->frontiered = false;
        // Erase the first element, pop it.
        frontier.erase(frontier.begin());

        // Add the current node to the explored set.
        explored.emplace_back(current_node);
        current_node->explored = true;

        // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
        Node* result_child = ActionSpace(current_node, frontier, StateMatrix, UCS, visualize_vector);
        if (result_child) {
            std::cout << "\nExplored Set:\n";
            VisualizeVector(explored);
            return result_child;
        }
    }
}
```

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

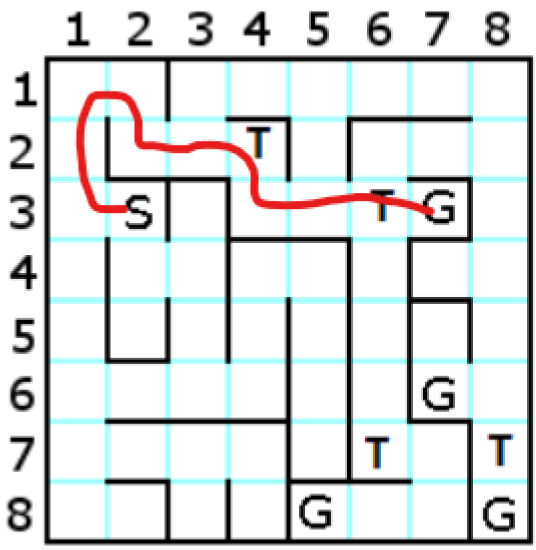
```cpp
    case UCS:
    {
        for (Node* cur_child : action_vector)
        {
            if (!cur_child->explored && !cur_child->frontiered) {
                cur_child->parent = current_node;
                if (cur_child->type == 'G') {
                    cur_child->explored = true;
                    return cur_child;
                }
                cur_child->frontiered = true;
                frontier.emplace_back(cur_child);
            }
        }
        std::sort(frontier.begin(), frontier.end(), CompareTwoNodesCosts);
        if (visualize_vector)
            VisualizeVector(frontier);
    }
    break;
```

After, addition of new nodes, list gets sorted.

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

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)
----------
```

```cpp
// Execution of GBFS
Node* ExecuteGBFS(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector) {

    std::cout << "GBFS is selected." << std::endl;
    while (true)
    {
        // If frontier is empty, there is something wrong.
        if (frontier.empty()) {
            std::cout << "Failure... Path didn't found.\n";
            return NULL;
        }
        // Get the current node from the first element of frontier.
        Node* current_node;
        current_node = frontier[0];
        current_node->frontiered = false;

        // Erase the first element, pop it.
        frontier.erase(frontier.begin());

        // Add the current node to the explored set.
        explored.emplace_back(current_node);
        current_node->explored = true;

        // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
        Node* result_child = ActionSpace(current_node, frontier, StateMatrix, GBFS, visualize_vector);
        std::cout << "\nExplored Set:\n";
        VisualizeVector(explored);
        if (result_child) {
            return result_child;
        }
    }
}
```

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

```cpp
case GBFS:
{
    for (Node* cur_child : action_vector)
    {
        if (!cur_child->explored && !cur_child->frontiered) {
            cur_child->parent = current_node;
            if (cur_child->type == 'G') {
                cur_child->explored = true;
                return cur_child;
            }
            cur_child->frontiered = true;
            frontier.emplace_back(cur_child);
        }
    }
    std::sort(frontier.begin(), frontier.end(), CompareTwoNodesHeuristics);
    if (visualize_vector)
        VisualizeVector(frontier);
}
break;
```

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

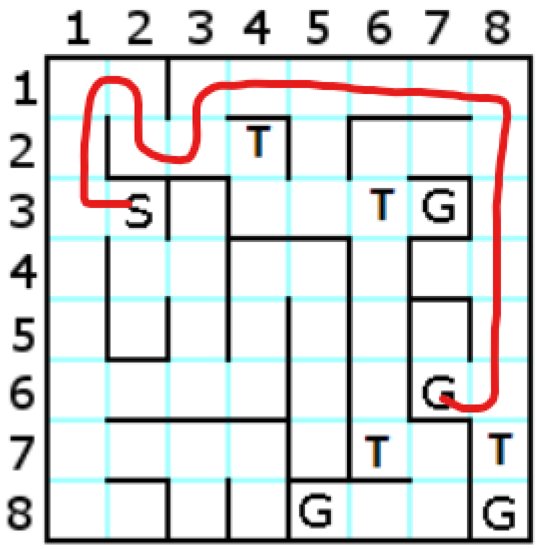
```cpp
// Compares two nodes with their heuristic path costs.
bool CompareTwoNodesHeuristics(Node* n1, Node* n2) {
    return CalculateHeuristic(n1) < CalculateHeuristic(n2);
}
```

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

```
// Calculating the Manhattan Distance Heuristic.
int CalculateHeuristic(Node* current_node) {

    if (current_node->type == 'G')
        return 0;

    int min_manhattan_distance = 9999999;

    for (std::size_t i = 0; i < GoalNodes.size(); i++)
    {
        int cur_distance = abs(GoalNodes[i]->x - current_node->x) + abs(GoalNodes[i]->y - current_node->y);
        if (cur_distance < min_manhattan_distance) {
            min_manhattan_distance = cur_distance;
        }
    }
    return min_manhattan_distance;
}
```

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: 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
```

```cpp
Node* ExecuteASTAR(std::vector<std::vector<Node*>> StateMatrix, std::vector<Node*> frontier, std::vector<Node*> explored, bool visualize_vector)
{
    std::cout << "ASTAR is selected." << std::endl;
    while (true)
    {
        // If frontier is empty, there is something wrong.
        if (frontier.empty()) {
            std::cout << "Failure... Path didn't found.\n";
            return NULL;
        }
        // Get the current node from the first element of frontier.
        Node* current_node;
        current_node = frontier[0];
        current_node->frontiered = false;

        // Erase the first element, pop it.
        frontier.erase(frontier.begin());

        // Add the current node to the explored set.
        explored.emplace_back(current_node);
        current_node->explored = true;

        // Send the current node to the ActionSpace, collect if it finds the goal node in this depth.
        Node* result_child = ActionSpace(current_node, frontier, StateMatrix, ASTAR, visualize_vector);
        std::cout << "\nExplored Set:\n";
        VisualizeVector(explored);
        if (result_child) {
            return result_child;
        }
    }
}
// Execution of IDS
```

Same with GBFS.

```cpp
    case ASTAR:
    {
        for (Node* cur_child : action_vector)
        {
            if (!cur_child->explored && !cur_child->frontiered) {
                cur_child->parent = current_node;
                if (cur_child->type == 'G') {
                    cur_child->explored = true;
                    return cur_child;
                }
                cur_child->frontiered = true;
                frontier.emplace_back(cur_child);
            }
        }
        std::sort(frontier.begin(), frontier.end(), CompareTwoNodesTotalCosts);
        if (visualize_vector)
            VisualizeVector(frontier);
    }
    break;
```

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

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

This time, current cost is added to the heuristics.

```cpp
int CurrentPathCost(Node* current_node) {

    int total_cost = 0;
    while (current_node)
    {
        total_cost += current_node->cost;
        current_node = current_node->parent;
    }
    return total_cost;
}
```

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