Maze Generator and solver

Team Members:

- Abdelrahman Ahmed Safwat 21-101108
- Doha Bahaa eldin 21-101136
- Kareem Abdelhameed 21-101015
- Hamdi Awad 21-101011

User Input:

The User inputs the maze size n. Where the maze is divided into a group of nodes or squares, where n denotes the number of squares per row or column, thus the size of the maze is n x n. The program then starts generating the maze and is prompted to choose a solving algorithm

Target Maze:

The target maze that our program generates is a maze of size n x n, where n is based on the user input. The start node is at the top right corner (node 0, 0) and the target or end is at the bottom right corner (node n - 1 , n - 1).

Important note: Future references to n during time complexity analysis refers to ***this n***, thus n^2 refers to rows * columns = n * n. It is NOT the number of vertices/nodes. But rather the number of vertices per row or per column

Maze Generation: **Abdelrahman Ahmed Safwat**
(Time complexity is always n^2 as it must visit all the nodes)
Maze Generation works via randomized DFS.
Coloring : To portray the DFS in action, visited nodes are colored Cyan, and then colored back white upon finishing.

```
colorNode(x, Color::Cyan);
colorNode(x, Color::White);
```

It starts at 0, 0, and performs the following operations over this node and all the other nodes :

1 - Choose a random neighbor to the current node in one of the 4 directions (North - South - East - West). The method of randomization works by saving these four directions in a vector, and randomly choosing an index and removing it afterwards.

```
vector<node> neighbours = surrounding(x);
int index = rand() % neighbours.size();
node neighbour = neighbours[index];
```

2 - Carve a tunnel from the current node to this chosen neighbor. Typically, a tunnel has two openings. In our case, the first is the current node, and the second is the chosen neighbor. Thus walls are built surrounding the current node with the other UNchosen neighbors. Wall building is done using the following code:

```
for (auto &otherNeighbour : otherNeighbours)
     if (otherNeighbour == neighbour) continue;
     if (visited.find(otherNeighbour) == visited.end())
          colorWall(x, otherNeighbour, Color::Black);
```

→ These walls can be removed later however, if the neighbor was found to be unvisited

3 - It gets connected to the current node in our graph, then is visited. This is done by the following code :

```
makeEdge(x, neighbour);
randomizedDfs(neighbour);
```

Graphical Interface: **Abdelrahman Ahmed Safwat**
The library used in the graphical interface is SFML. Every single element drawn on screen is a rectangle, it is all rectangle manipulations. For instance, walls are thin rectangles. The white background is a rectangle with equal sides (Basically a square). The individual nodes are also squares.

The Maze window is 860 x 860 pixels. 800 dedicated for the nodes, where each node takes 800 / n x 800 / n pixels. And 60 dedicated to the walls, where each wall also takes 800 / n pixels for the long side and 60 / n for the thin side.

```
RECT_SIZE = 800.0 / SIZE;
WALL_SIZE = 60.0 / (SIZE - 1); // - 1 for integer division
```

It keeps track of nodes and walls in two separate arrays of rectangles. Any color changes are primarily through these arrays.

```
vector<vector<RectangleShape>> grid;
vector<RectangleShape> walls;
```

This part of the code includes several functions, for printing, clearing the screen, …etc. Each function goes over these two arrays and draws these rectangles to the screen after any change.

Maze solving: DFS: **Hamdi Awad**

(Time complexity worst case n^2 → visits all nodes)

Dfs works by visiting a node, and then exploring its different neighbors via recursion. Similar to Maze Generation, it colors the visited node Green, rather than Cyan.

```
colorNode(S, Color::Green);
visitedDFS.insert(S);
```

A key difference is that the DFS solving algorithm wants to maintain the solution path, rather than changing all nodes back to white. Thus the DFS function returns a boolean, indicating whether or not this node is on the solution path, and if it is, it returns true, thus leaving it green and stopping exploration to the rest of the neighbors.

```
if (S.x == SIZE - 1 && S.y == SIZE - 1)
     visitedDFS.clear();
     return true;

for (auto &i : neighbours(S))
     if (visitedDFS.find(i) == visitedDFS.end())
          if (dfs(i))
               return true;
```

Otherwise, it colors it back white and returns false.

```
colorNode(S, Color::White);
return false;
```

Maze solving: BFS: **Doha Bahaa eldin**

(Time complexity worst case n^2 → visits all nodes, but in practice it is ***much better*** than n^2 on average thus much faster than DFS because the BFS is able to get the shortest path between any pair of nodes in an unweighted graph)

In the BFS Maze Solver, We traverse over each layer of available nodes till we reach the end node, that's when we know that we found our shortest path so we go back to the traced parents along our traversal and print the chosen path.

1 - We initialize a set of visited nodes and a queue of nodes for the BFS traversal, we mark the starting node as visited and push it into the queue, we continue until the queue is empty, storing the first element in queue and then removing it and coloring the node which indicates that it has been visited.

```
set<node> visited;
queue<node> Q;
visited.insert(S);
Q.push(S);

while (!Q.empty())
      node temp = Q.front();
      Q.pop();
      colorNode(temp, Color::Cyan);
```

2 - If the destination is reached, store the destination node and exit the loop

```
if (temp.y == SIZE - 1 && temp.x == SIZE - 1)
      X = temp;
      break;
```

3 - We store the neighbors of the current node in a vector called N, then we traverse all neighbors, if the neighbor is not visited, we update the parent of neighbor, push neighbor into the queue and mark as visited

```
vector<node> N = neighbours(temp);

for (auto neighbour : N)
     if (visited.find(neighbour) == visited.end())
          parent[neighbour] = temp;
          Q.push(neighbour);
          visited.insert(neighbour);
```

4 - For visualization we trace the path from the destination to the source. And we Visualize the path in whatever color we want then move to the parent node and finally we visualize the source node in the same color (purple)

```
while (!(X == S))
     colorNode(X, Color(160, 32, 240));
     X = parent[X];

colorNode(X, Color(160, 32, 240));
```

Maze solving: A*: **Kareem AbdelHameed**
(Time complexity worst case n^2 → visits all nodes. Similarly to BFS, in practice it is ***much better*** than n^2 on average. A* is a modification for dijkstra's algorithm, and in an unweighted graph as is our case, dijkstra would be identical BFS, but given the heuristic modification, it can actually surpass the BFS and reach the goal faster according to the chosen manhattan distance)

In the A* technique we determine the shortest path, just like a special form of dijkstra's algorithm. The nodes are selected based on their current distance from the source, In addition to their Manhattan distance from the target. The main loop continues until the target is reached. In each iteration, the algorithm explores neighboring nodes, updating distances based solely on the actual cost from the source.

1 - We initialize distances to all nodes as infinity, while the distance from the source to itself as 0 and mark the source node as visited.

```
for (int i = 0; i < SIZE; i++)
     for (int z = 0; z < SIZE; z++)
          node x(i, z);
          dis[x] = INT_MAX;

dis[source] = 0;
```

2 - We initialize a priority queue with pairs (priority, node), and we push the source node onto the priority queue with priority 0.

```
priority_queue<pair<pair<int, int>, node>> q;
q.push({{0, 0}, source});
```

3 - Main loop: we continue until the queue is empty, retrieving the node with the highest priority, and removing it, marking the current node as visited

```
while (!q.empty())
     pair<pair<int, int>, node> temp = q.top();
     q.pop();
     visited.insert(temp.second);
```

4 - If the destination is reached, store the destination node and exit the loop

```
if (temp.second.y == SIZE - 1 && temp.second.x == SIZE - 1)
      X = temp.second;
      break;
```

5 - For all neighbors, Check if a shorter path to the neighbor is found and the neighbor is not visited. We update the distance to the neighbor, Push the neighbor onto the priority queue with updated priority and distance and update the parent of the neighbor.

```
for (auto neighbor : neighbours(temp.second))

      if (dis[temp.second] + 1 < dis[neighbor] &&
      visited.find(neighbor) == visited.end())

            dis[neighbor] = dis[temp.second] + 1;
            int manhaten = SIZE - 1 - neighbor.x + SIZE - 1 -neighbor.y;
            q.push({{dis[neighbor] +manhaten, dis[neighbor]},neighbor});
            parent[neighbor] = temp.second;

//end of main loop. //next is just the visualization, and how we color
our steps to form a maze identical to the previous algorithm.
```