**Reference Repository:** [**https://github.com/ravenkls/Maze-Generator-and-Solver**](https://github.com/ravenkls/Maze-Generator-and-Solver)

Files and Functions:

**Analyze\_efficieny.py**: analyses the efficiency of the maze generation algorithm.

**Djikstra.py:**

1. PriorityItem:
   * Description: Represents an item with a priority (used for inputting non-comparable data into a priority queue).
   * Purpose: Used in Dijkstra’s algorithm for maintaining the priority queue.
2. run\_dijkstra\_algorithm(start\_node, nodes):
   * Description: Executes Dijkstra’s algorithm on an array of nodes, given a starting node.
   * Updates all nodes with a ‘distance’ value (shortest distance from the start node).
   * Updates all nodes with a ‘previous’ value (previous node in the shortest path).
3. get\_path\_from\_node(node: Node) -> List[Node]:
   * Description: Returns the list of nodes leading from a specific node to the start node.
   * Purpose: Used to extract the shortest path from Dijkstra’s algorithm results.
4. colour\_path(image, path):
   * Description: Colors in a path (based on nodes) from red to green in the maze image.
   * Purpose: Visualizes the solved path.
5. solve\_image(file\_path) -> None:
   * Description: Solves a maze image file and outputs the solution in the same folder.
   * Steps:
     + Reads the image.
     + Extracts start and finish nodes from the maze.
     + Runs Dijkstra’s algorithm.
     + Retrieves the shortest path.
     + Colors the path in the image.
     + Saves the solved image.

**Maze\_analyzer.py:**

1. Node:
   * Description: Represents a node on a weighted graph for a maze.
   * Attributes:
     + coords: Coordinates of the node.
     + previous: Reference to the previous node in the shortest path.
     + distance: Shortest distance from the start node.
     + adjacency\_list: Dictionary of adjacent nodes and their weights.
   * Methods:
     + link(node, weight=1): Links two nodes together with a specified weight.
     + \_\_repr\_\_(): Returns a string representation of the node.
2. nodes\_from\_maze(image) -> (Node, Node, List[Node]):
   * Description: Generates a list of nodes for a maze, linking them with weighted edges based on their distance from each other.
   * Steps:
     + Reads the maze image.
     + Identifies start and finish nodes.
     + Creates nodes for turning points or dead ends.
     + Establishes horizontal and vertical paths.
     + Converts nodes dictionary into a list.
   * Returns:
     + start\_node: Start node of the maze.
     + finish\_node: Finish node of the maze.
     + nodes: List of all nodes in the maze.

**Maze\_generator.py:**

1. generate\_maze(width, height):
   * Description: Generates a maze image using the recursive backtracker algorithm.
   * Steps:
     + Creates an image (maze) with dimensions (2\*width + 1, 2\*height + 1) and initializes it with black pixels.
     + Sets the entrance (top) and exit (bottom) paths by changing specific pixels to white.
     + Initializes a stack (LifoQueue) and a 2D array (cells) to track visited cells.
     + Starts from the top-left corner (cell (0, 0)).
     + Randomly selects unvisited adjacent cells.
     + Removes walls between the current cell and the chosen neighbor.
     + Updates the cells array to mark visited cells.
     + Continues until all cells are visited.
   * Returns: The generated maze as an image.
2. if \_\_name\_\_ == '\_\_main\_\_'::
   * Description: Entry point for executing the script.
   * Parses command-line arguments (maze width, height, and output file path).
   * Calls generate\_maze with specified dimensions.
   * Saves the generated maze image to the specified output file.