REPORT OF THE PROJECT

INTRODUCTION

This project, titled Gold Trail: The Knight's Path, aims to implement a grid-based pathfinding system where a knight navigates a terrain to collect gold coins. The map is composed of various terrain types (grass, sand, and impassable obstacles) each associated with different movement costs. The knight must travel from a given starting location to a sequence of coin positions while minimizing the total travel cost.

The implementation is developed using the Java programming language in an object-oriented way. The program reads terrain and cost information from structured input files and applies Dijkstra's algorithm to find the shortest path to each objective in sequence. Visualization is handled by the StdDraw graphics library, which animates the environment, knight's movement, and collected objectives if the user enables the optional -draw flag. *(in Fİgure-1)*

Throughout the simulation, the knight moves in one of four directions (up, down, left, right), and only passable tiles are considered in path computation. At each successful reach of an objective, the knight's position is updated, and the next target becomes the new destination. The program also generates a summary file, output.txt *(in Figure-2)*, detailing the steps taken and the total cost of each path.

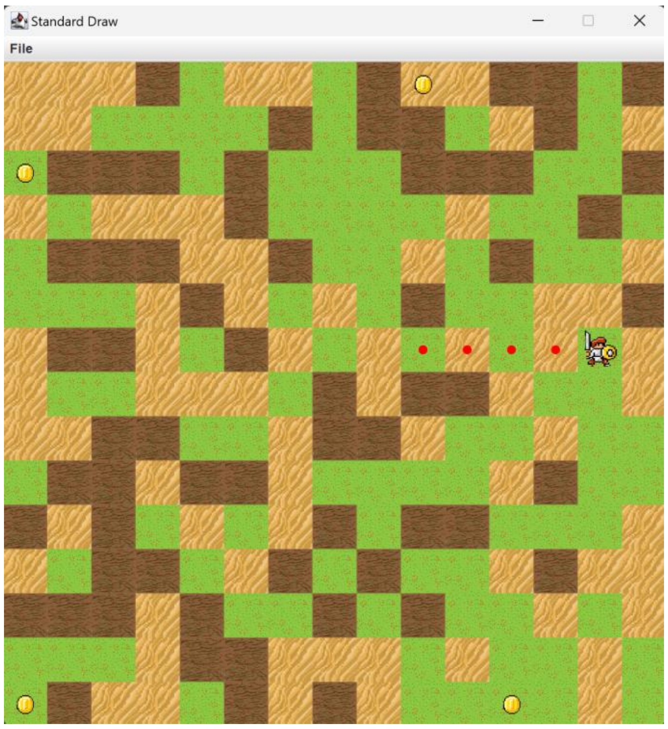
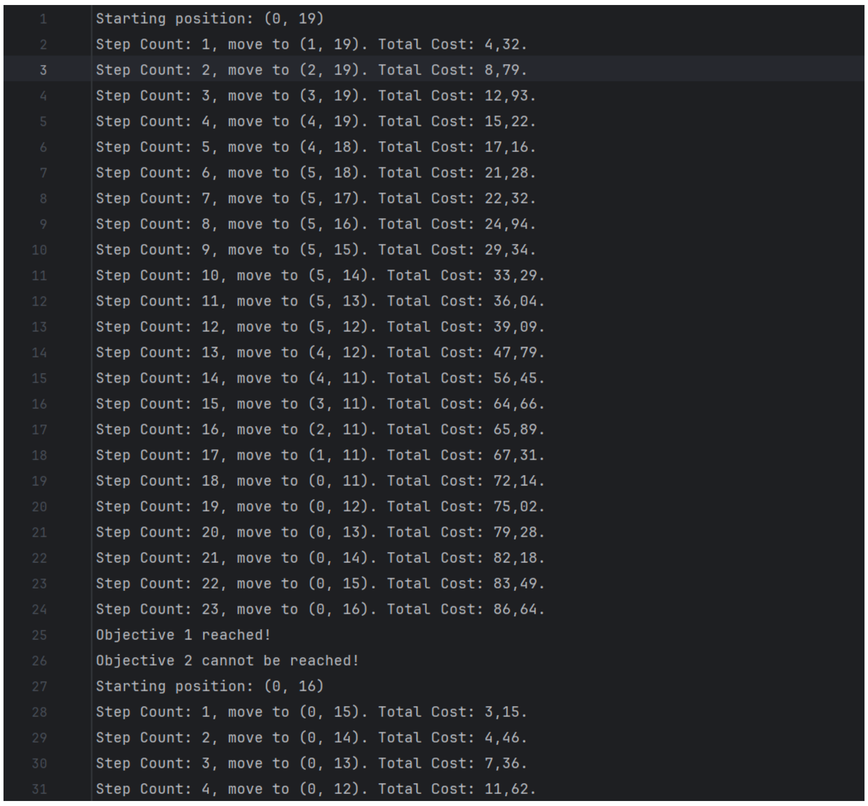


Figure-1



```
Starting position: (0, 19)
Step Count: 1, move to (1, 19). Total Cost: 4,32.
Step Count: 2, move to (2, 19). Total Cost: 8,79.
Step Count: 3, move to (3, 19). Total Cost: 12,93.
Step Count: 4, move to (4, 19). Total Cost: 15,22.
Step Count: 5, move to (4, 18). Total Cost: 17,16.
Step Count: 6, move to (5, 18). Total Cost: 21,28.
Step Count: 7, move to (5, 17). Total Cost: 22,32.
Step Count: 8, move to (5, 16). Total Cost: 24,94.
Step Count: 9, move to (5, 15). Total Cost: 29,34.
Step Count: 10, move to (5, 14). Total Cost: 33,29.
Step Count: 11, move to (5, 13). Total Cost: 36,04.
Step Count: 12, move to (5, 12). Total Cost: 39,09.
Step Count: 13, move to (4, 12). Total Cost: 47,79.
Step Count: 14, move to (4, 11). Total Cost: 56,45.
Step Count: 15, move to (3, 11). Total Cost: 64,66.
Step Count: 16, move to (2, 11). Total Cost: 65,89.
Step Count: 17, move to (1, 11). Total Cost: 67,31.
Step Count: 18, move to (0, 11). Total Cost: 72,14.
Step Count: 19, move to (0, 12). Total Cost: 75,02.
Step Count: 20, move to (0, 13). Total Cost: 79,28.
Step Count: 21, move to (0, 14). Total Cost: 82,18.
Step Count: 22, move to (0, 15). Total Cost: 83,49.
Step Count: 23, move to (0, 16). Total Cost: 86,64.
Objective 1 reached!
Objective 2 cannot be reached!
Starting position: (0, 16)
Step Count: 1, move to (0, 15). Total Cost: 3,15.
Step Count: 2, move to (0, 14). Total Cost: 4,46.
Step Count: 3, move to (0, 13). Total Cost: 7,36.
Step Count: 4, move to (0, 12). Total Cost: 11,62.
```

Figure-2

CLASS DIAGRAMS

🔗 Relationships Between Classes

The project is structured around three main classes that collaborate to achieve the shortest pathfinding functionality:

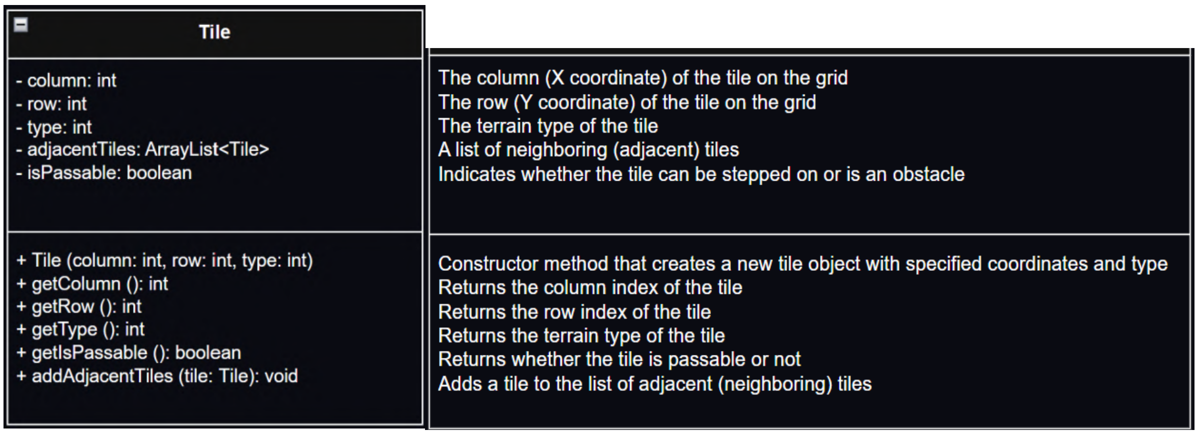
- **`Main` Class**: Acts as the central controller. It reads input files, initializes the map as a grid of `Tile` objects, and invokes the `PathFinder` to calculate optimal routes between objectives. It also handles visualization via `StdDraw`.

- **`Tile` Class**: Represents each individual cell on the map. It stores the tile's coordinates, type (grass, sand, or obstacle), passability status, and references to adjacent tiles.

- **`PathFinder` Class**: Uses the 2D grid of `Tile` objects and a 3D travel cost structure to find the shortest path between two points using **Dijkstra's algorithm**.

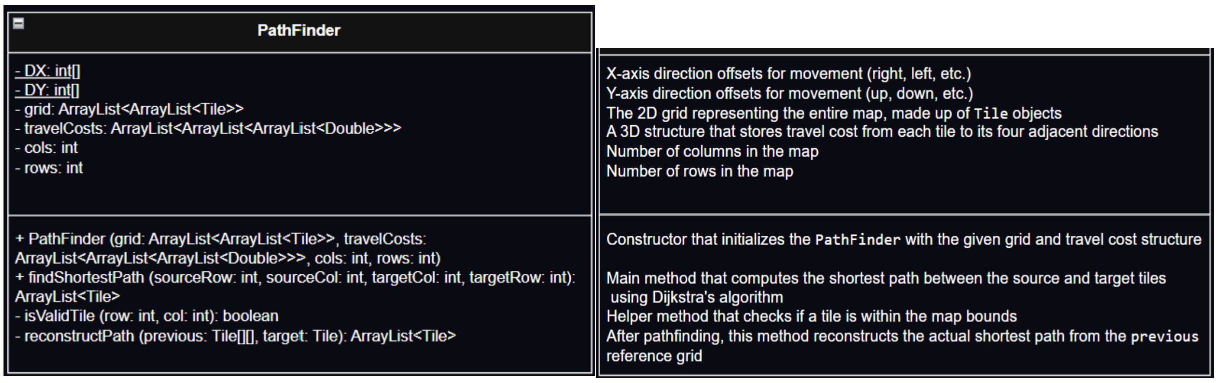
These classes interact in a coordinated manner where `Main` oversees the execution flow and visualization, `Tile` models the map's structure, and `PathFinder` provides the pathfinding logic based on terrain costs.

➔ UML Diagram of Tile Class



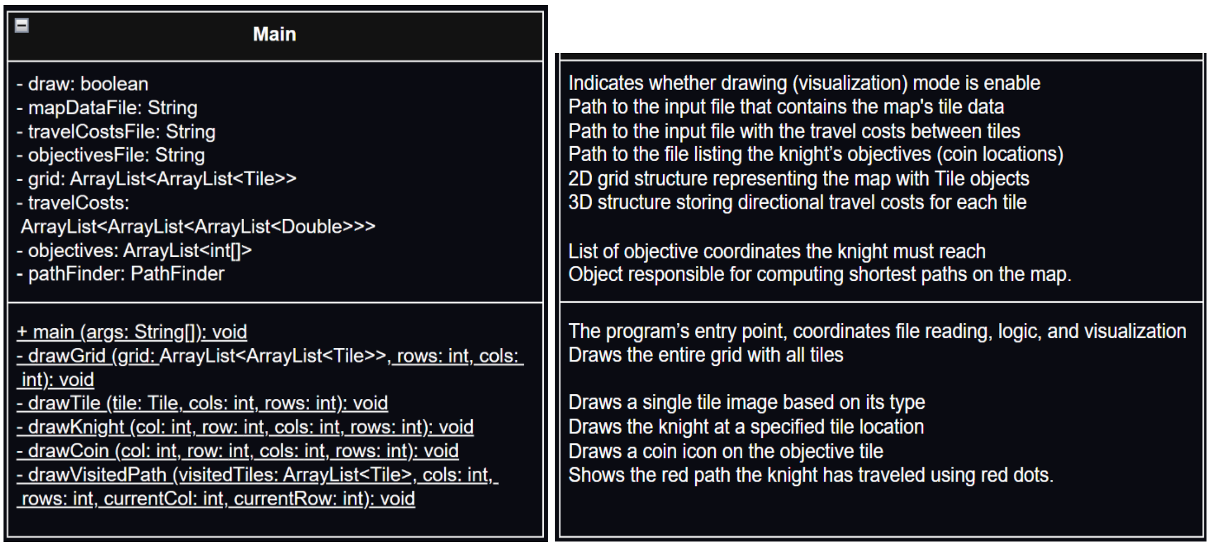
Tile	
- column: int	The column (X coordinate) of the tile on the grid
- row: int	The row (Y coordinate) of the tile on the grid
- type: int	The terrain type of the tile
- adjacentTiles: ArrayList<Tile>	A list of neighboring (adjacent) tiles
- isPassable: boolean	Indicates whether the tile can be stepped on or is an obstacle
+ Tile (column: int, row: int, type: int)	Constructor method that creates a new tile object with specified coordinates and type
+ getColumn (): int	Returns the column index of the tile
+ getRow (): int	Returns the row index of the tile
+ getType (): int	Returns the terrain type of the tile
+ getIsPassable (): boolean	Returns whether the tile is passable or not
+ addAdjacentTiles (tile: Tile): void	Adds a tile to the list of adjacent (neighboring) tiles

➔ UML Diagram of PathFinder Class



PathFinder	
- DX: int[]	X-axis direction offsets for movement (right, left, etc.)
- DY: int[]	Y-axis direction offsets for movement (up, down, etc.)
- grid: ArrayList<ArrayList<Tile>>	The 2D grid representing the entire map, made up of `Tile` objects
- travelCosts: ArrayList<ArrayList<ArrayList<Double>>>	A 3D structure that stores travel cost from each tile to its four adjacent directions
- cols: int	Number of columns in the map
- rows: int	Number of rows in the map
+ PathFinder (grid: ArrayList<ArrayList<Tile>>, travelCosts: ArrayList<ArrayList<ArrayList<Double>>>, cols: int, rows: int)	Constructor that initializes the `PathFinder` with the given grid and travel cost structure
+ findShortestPath (sourceRow: int, sourceCol: int, targetCol: int, targetRow: int): ArrayList<Tile>	Main method that computes the shortest path between the source and target tiles using Dijkstra's algorithm
- isValidTile (row: int, col: int): boolean	Helper method that checks if a tile is within the map bounds
- reconstructPath (previous: Tile[][], target: Tile): ArrayList<Tile>	After pathfinding, this method reconstructs the actual shortest path from the `previous` reference grid

➔ UML Diagram of Main Class



Main	
- draw: boolean	Indicates whether drawing (visualization) mode is enable
- mapDataFile: String	Path to the input file that contains the map's tile data
- travelCostsFile: String	Path to the input file with the travel costs between tiles
- objectivesFile: String	Path to the file listing the knight's objectives (coin locations)
- grid: ArrayList<ArrayList<Tile>>	2D grid structure representing the map with Tile objects
- travelCosts: ArrayList<ArrayList<ArrayList<Double>>>	3D structure storing directional travel costs for each tile
- objectives: ArrayList<int[]>	List of objective coordinates the knight must reach
- pathFinder: PathFinder	Object responsible for computing shortest paths on the map.
+ main (args: String[]): void	The program's entry point, coordinates file reading, logic, and visualization
- drawGrid (grid: ArrayList<ArrayList<Tile>>, rows: int, cols: int): void	Draws the entire grid with all tiles
- drawTile (tile: Tile, cols: int, rows: int): void	Draws a single tile image based on its type
- drawKnight (col: int, row: int, cols: int, rows: int): void	Draws the knight at a specified tile location
- drawCoin (col: int, row: int, cols: int, rows: int): void	Draws a coin icon on the objective tile
- drawVisitedPath (visitedTiles: ArrayList<Tile>, cols: int, rows: int, currentCol: int, currentRow: int): void	Shows the red path the knight has traveled using red dots.

ALGORITHM

The core of this project's pathfinding logic is built on Dijkstra's Algorithm, a well-known graph traversal algorithm designed to compute the shortest path in a weighted graph with non-negative edge weights. In this project, the grid is treated as a weighted graph where each tile is a node and adjacent passable tiles are connected by edges with weights determined by the terrain-specific movement cost.

1) Grid Representation and Costs

Each tile on the grid is modeled using the Tile class, which stores its type (grass, sand, obstacle), coordinates, passability status, and adjacent tiles. Movement costs between adjacent tiles are stored in a 3D list named travelCosts, where each cell contains a list of four directional costs corresponding to movement in the up, right, down, and left directions.

2) Path Computation

The PathFinder class is responsible for computing the shortest path between the knight's current location and a given objective. The algorithm follows these key steps:

➔ Initialization:

A 2D distances array is initialized with infinity (∞), representing the minimum cost to reach each tile. The starting tile's distance is set to 0. An auxiliary previous array stores the preceding tile for path reconstruction.

➔ Main Loop:

The algorithm iteratively selects the unvisited tile with the smallest current distance. It then examines all valid, unvisited adjacent tiles and calculates tentative distances through the current tile. If a shorter path is found, the distance and previous references are updated accordingly.

➔ Termination:

The loop continues until the target tile is visited or all reachable tiles are exhausted. If the target is unreachable, the method returns null.

➔ Path Reconstruction:

Once the target is reached, the algorithm traces back the path using the previous array and constructs the shortest path as a list of Tile objects.

3) Objective Sequencing

The Main class handles the overall program flow. After parsing input files, the knight is initialized at the first coordinate in objectives.txt. The program then processes each objective in sequence. If a path to an objective cannot be found (e.g., due to surrounding obstacles), the program writes this event to the output file as seen *in Figure-2* and continues with the next objective using the current position.