CmpE 160 Assignment 3
Gold Trail: The Knight’s Path

Abdullah Tuğra Acar
Student ID: 2023400219

May 11, 2025

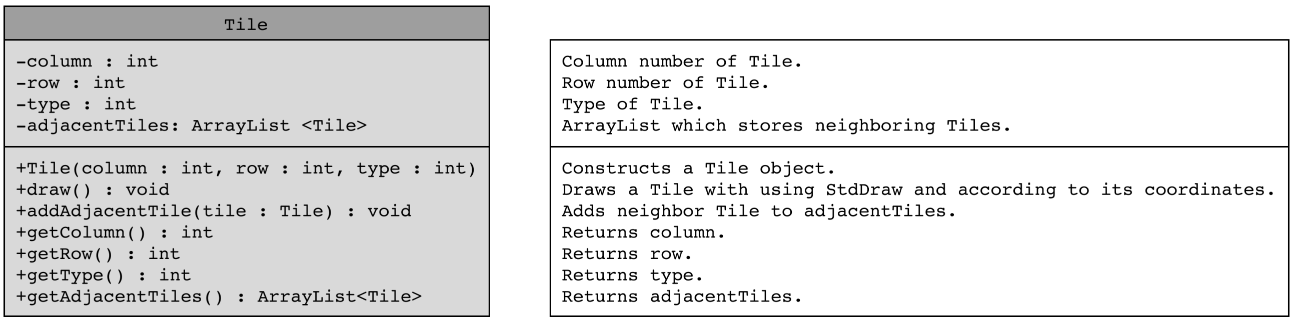
1 Introduction

In this project, we wrote an application that reads which tiles the map consists of with the map data given to us, reads the cost of transition between these tiles with the cost data, reads the location of the gold we need to find with the data of the objects, finds the shortest distance between the gold with this information, prints this path to a txt file and visualizes it using the StdDraw library. We used Dijkstra as the path finding algorithm, in the visualization, we scaled the canvas according to the size of the map, we put the tiles according to the tile coordinates to be 30*30, and when the knight found a path, we took the knight on the road between two gold coins in a way that would leave a trace where he passed. If there is no way to a gold, we said that we could not reach it without any attempt. Our program works with command line arguments, if you add the draw flag, it also does the visualization, otherwise it just finds the path and adds it to the txt file.

2 Class Diagrams

2.1 Tile Class

The ‘Tile‘ class defines a single square on a grid-based map. It holds the tile’s position using ‘row‘ and ‘column‘ values, as well as its terrain type as ints (e.g., grass(0), sand(1), or impassable(2)). Each tile also stores a list of adjacent, reachable tiles, which helps in determining movement options. The ‘draw()‘ method displays the tile visually using the StdDraw library. Overall, the class helps structure the map and supports basic navigation between connected tiles.



Tile	
-column : int	Column number of Tile.
-row : int	Row number of Tile.
-type : int	Type of Tile.
-adjacentTiles: ArrayList <Tile>	ArrayList which stores neighboring Tiles.
+Tile(column : int, row : int, type : int)	Constructs a Tile object.
+draw() : void	Draws a Tile with using StdDraw and according to its coordinates.
+addAdjacentTile(tile : Tile) : void	Adds neighbor Tile to adjacentTiles.
+getColumn() : int	Returns column.
+getRow() : int	Returns row.
+getType() : int	Returns type.
+getAdjacentTiles() : ArrayList<Tile>	Returns adjacentTiles.

Figure 1: UML Diagram of the Tile class.

2.2 Map Class

The Map class represents a grid-based environment composed of multiple Tile objects. It reads map data from a file to initialize the grid's dimensions and collect each tile. The class initialize passible neighbors of tiles, excludes impassable ones. With getters you can access to a specific tile, dimensions of map, and the entire tile array. The draw() method visually renders the full map and any remaining coins using the StdDraw library.

Map	
-row : int -column : int -tiles : Tile[]	How many columns of Map. How many rows of Map. List of Tiles in Map.
+Map(mapData : File) throws Exception +getTileAt(x : int, y : int) : Tile +getTiles() : Tile[] +getColumn() : int +getRow() : int +draw(coins : List<int[]>) : void	Constructs a Map object. Returns Tile at given positions. Returns tiles list of that Map. Returns column number. Returns row number. Draws the map according to given coins information.

Figure 2: UML Diagram of the Map class.

2.3 Cost Class

The Cost class handles movement costs between tiles by reading data from a file and storing it in a HashMap. File has it one way but it adds both ways. It allows finding cost of a travel with coordinates of start and goal tiles.

Cost	
-costMap : HashMap<String, Double>	HashMap of costs between two Tile.
+Cost(travelCosts : File) throws Exception +getCost(from : Tile, to : Tile) : Double	Constructs a Cost object. Returns cost between initial and goal Tiles.

Figure 3: UML Diagram of the Cost class.

2.4 PathFinder Class

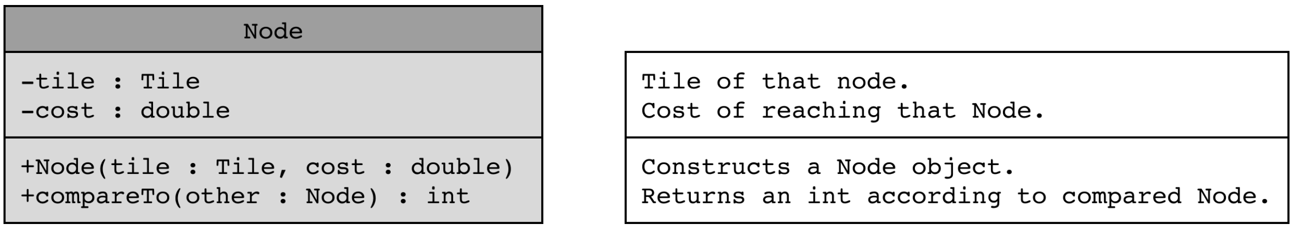
The PathFinder class is responsible for finding the shortest path between two tiles on the map. It implements Dijkstra's algorithm, using a priority queue to explore tiles based on the lowest total travel cost. Movement costs between tiles are from the Cost class, and possible travelings are determined by using adjacent tiles from the Map. The algorithm tracks visited tiles and previous steps to reconstruct the optimal path once the goal is reached. The resulting path is returned as an ordered list of tiles representing the steps to follow.

PathFinder	
-map : Map -costGraph : Cost	Map object to find the path. Cost object to find the path.
+PathFinder(map : Map, costGraph : Cost) +findPath(start : Tile, goal : Tile) : List<Tile> -reconstructPath(previous : HashMap<Tile, Tile>, goal : Tile) : List<Tile>	Constructs a PathFinder object. Path finding algorithm which uses Dijkstra. A private function that helps to Dijkstra if the tile is reached trying to find another path.

Figure 4: UML Diagram of the PathFinder class.

2.4.1 Node Class

The Node class is a helper class used within the PathFinder to represent a tile along with its current path cost. Each Node stores a reference to a Tile and the total cost to reach it. It uses Double.compare to handle nodes are sorted by cost in the priority queue.



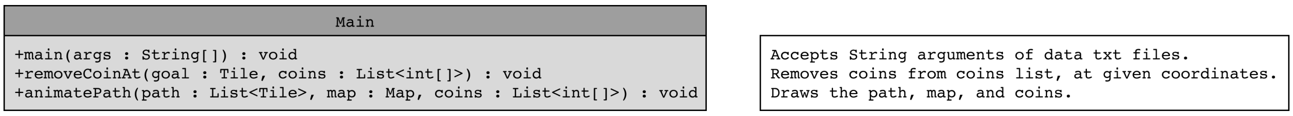
```
Node
-tile : Tile
-cost : double
+Node(tile : Tile, cost : double)
+compareTo(other : Node) : int
```

```
Tile of that node.
Cost of reaching that Node.
Constructs a Node object.
Returns an int according to compared Node.
```

Figure 5: UML Diagram of the Node class.

2.5 Main Class

The Main class is the central controller of the application, coordinating file input, pathfinding, visualisation and output. It starts by parsing command line arguments, optionally enabling animation with the -draw flag. It reads input files to construct the Map, loads movement costs into the Cost class, and retrieves a list of objectives (and coins) to collect. Using the PathFinder class, it calculates the shortest path between each reachable target, starting from an initial tile. As each path is found, the program writes step-by-step movement details and accumulated costs to an output file. When enabled, the animatePath() method visualises each path using the StdDraw library, showing the knight's movement and remaining coins. Helper methods like removeCoinAt() manage the coin state. The class also keeps track of the total number of steps and the total cost of the journey at the end.



```
Main
+main(args : String[]) : void
+removeCoinAt(goal : Tile, coins : List<int[]>) : void
+animatePath(path : List<Tile>, map : Map, coins : List<int[]>) : void
```

```
Accepts String arguments of data txt files.
Removes coins from coins list, at given coordinates.
Draws the path, map, and coins.
```

Figure 6: UML Diagram of the Main class.

3 Algorithm

The program uses a path-finding algorithm to find coins on a two-dimensional map. It uses the Dijikstra algorithm, which allows us to efficiently find the shortest path between two blocks. This is done by keeping each tile object as a ¡tile,cost¿ node and finding the shortest path from the first selected tile to the target tile. If the cost of the path used to reach a tile is less than the cost in the current node, the cost value is updated. Now let's examine the general algorithm of the program step by step.

Step-by-Step Process

1. **Initialization:** Before initialization begins, the program checks the command-line arguments to determine whether drawing mode is enabled. If the first argument is `-draw`, the program sets the global `DRAW` flag to `true` and adjusts the index for reading input files accordingly with setting `int entry` variable to 1. This enables visual animation of the knight's movement using the `StdDraw` library. Then, the `Main` class reads the input files:

- mapData.txt to initialize tile grid structure (`Map` class).
- travelCosts.txt to store movement costs between tile pairs (`Cost` class).
- objectives.txt to determine start point and coin locations.
- These txt files are all command line arguments. Our program accept these as:

```
java -cp "out:localPath/stdlib.jar" Main -draw mapData.txt
                    travelCosts.txt objectives.txt
```

2. **Map Construction:** The `Map` class creates a list of `Tile` objects and with `initializeAdjacentTiles` it links each tile to its valid, passable adjacent neighbors, forming a tree structure.

3. **Costs and Costs HashMap:** The `Cost` class uses a `HashMap` to store and retrieve movement costs between any two adjacent tiles. Even the doc has a cost between two cells just once its algorithm adds it in both way.

4. **Pathfinding:** The `PathFinder` class runs **Dijkstra's algorithm** to compute the shortest path from the current tile to the next objective. A priority queue is used to explore the lowest-cost tile at each step. Once the goal is reached, the path is reconstructed and returned as a list of `Tile` objects.

5. **Visualization (Optional):** If the `-draw` flag is provided, the `Main` class uses the `StdDraw` library to animate the knight's movement and dynamically update the remaining coin positions using the `animatePath()` method.

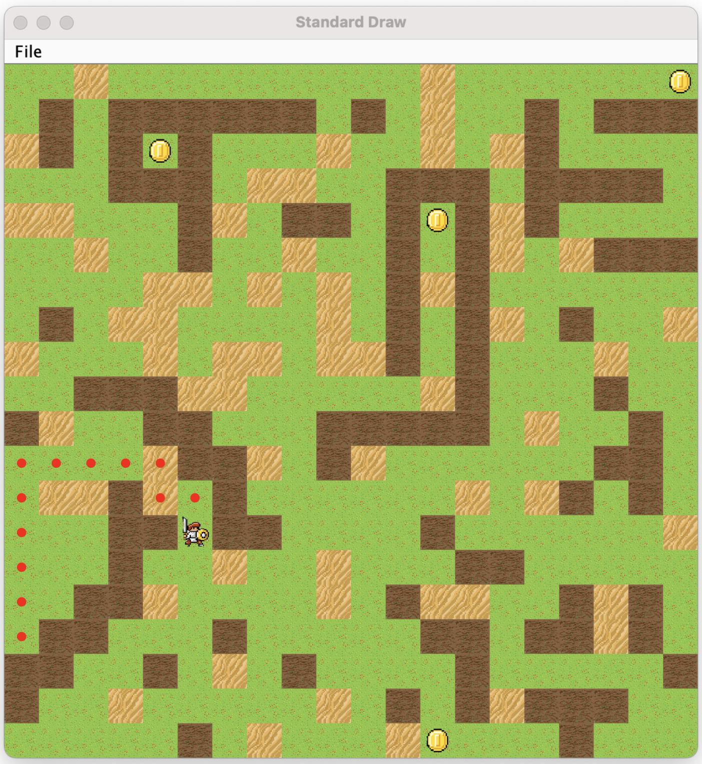
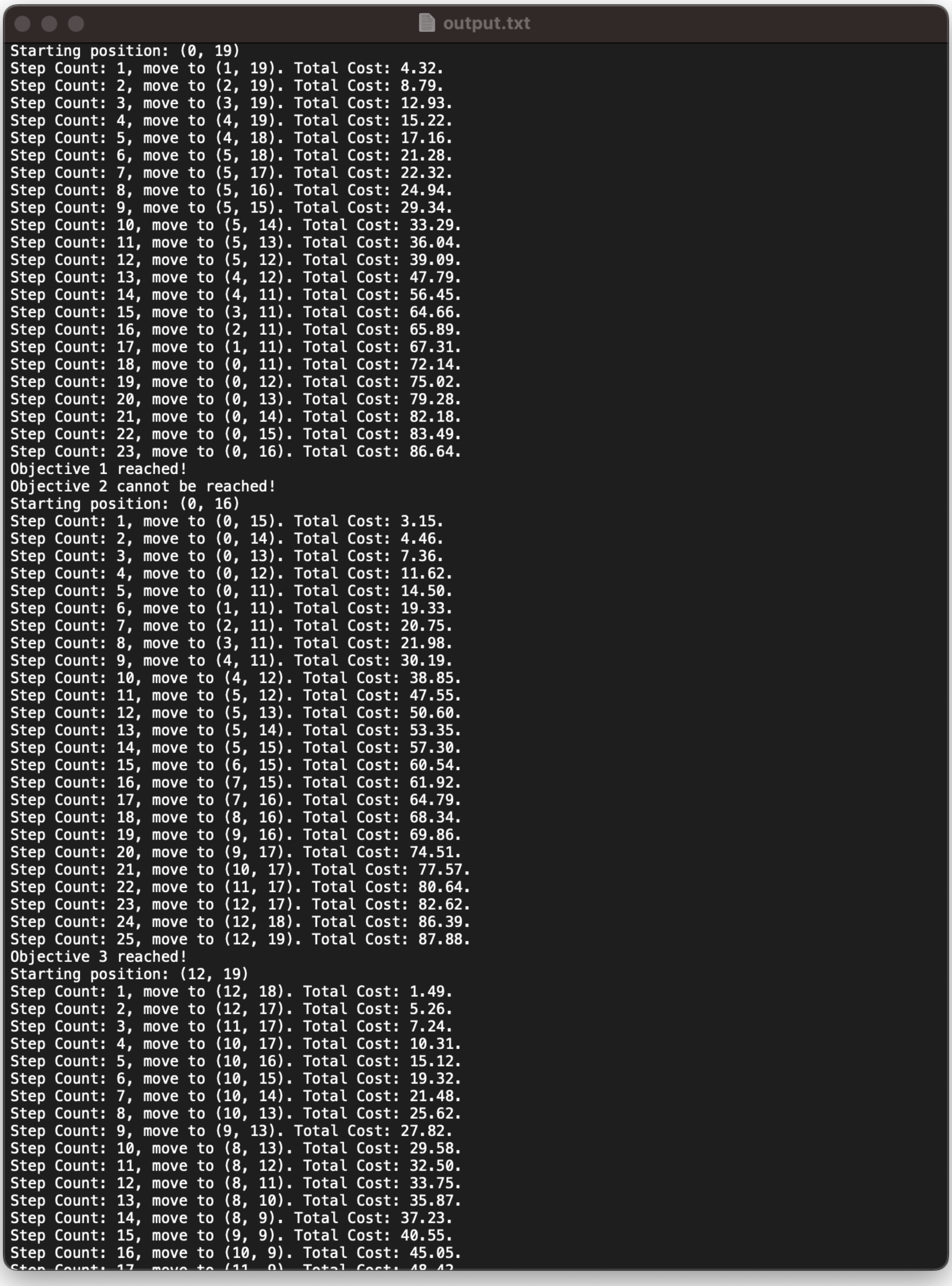


Figure 7: A drawing sample from the program with 20*20 map. (when draw flag is used).

6. **Output:** For each path, step-by-step directions and cumulative costs are written to an output file. The total number of steps and the overall cost are also reported at the end.



```
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.
Step Count: 5, move to (0, 11). Total Cost: 14.50.
Step Count: 6, move to (1, 11). Total Cost: 19.33.
Step Count: 7, move to (2, 11). Total Cost: 20.75.
Step Count: 8, move to (3, 11). Total Cost: 21.98.
Step Count: 9, move to (4, 11). Total Cost: 30.19.
Step Count: 10, move to (4, 12). Total Cost: 38.85.
Step Count: 11, move to (5, 12). Total Cost: 47.55.
Step Count: 12, move to (5, 13). Total Cost: 50.60.
Step Count: 13, move to (5, 14). Total Cost: 53.35.
Step Count: 14, move to (5, 15). Total Cost: 57.30.
Step Count: 15, move to (6, 15). Total Cost: 60.54.
Step Count: 16, move to (7, 15). Total Cost: 61.92.
Step Count: 17, move to (7, 16). Total Cost: 64.79.
Step Count: 18, move to (8, 16). Total Cost: 68.34.
Step Count: 19, move to (9, 16). Total Cost: 69.86.
Step Count: 20, move to (9, 17). Total Cost: 74.51.
Step Count: 21, move to (10, 17). Total Cost: 77.57.
Step Count: 22, move to (11, 17). Total Cost: 80.64.
Step Count: 23, move to (12, 17). Total Cost: 82.62.
Step Count: 24, move to (12, 18). Total Cost: 86.39.
Step Count: 25, move to (12, 19). Total Cost: 87.88.
Objective 3 reached!
Starting position: (12, 19)
Step Count: 1, move to (12, 18). Total Cost: 1.49.
Step Count: 2, move to (12, 17). Total Cost: 5.26.
Step Count: 3, move to (11, 17). Total Cost: 7.24.
Step Count: 4, move to (10, 17). Total Cost: 10.31.
Step Count: 5, move to (10, 16). Total Cost: 15.12.
Step Count: 6, move to (10, 15). Total Cost: 19.32.
Step Count: 7, move to (10, 14). Total Cost: 21.48.
Step Count: 8, move to (10, 13). Total Cost: 25.62.
Step Count: 9, move to (9, 13). Total Cost: 27.82.
Step Count: 10, move to (8, 13). Total Cost: 29.58.
Step Count: 11, move to (8, 12). Total Cost: 32.50.
Step Count: 12, move to (8, 11). Total Cost: 33.75.
Step Count: 13, move to (8, 10). Total Cost: 35.87.
Step Count: 14, move to (8, 9). Total Cost: 37.23.
Step Count: 15, move to (9, 9). Total Cost: 40.55.
Step Count: 16, move to (10, 9). Total Cost: 45.05.
Step Count: 17, move to (11, 9). Total Cost: 48.42.
```

Figure 8: A output.txt sequence sample from the program with 20*20 map.