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

Contact:
Ömer Ozan Mart (omer.mart@std.bogazici.edu.tr)
Deniz Baran Aksoy (deniz.aksoy@bogazici.edu.tr)

Due: May 12th 5 a.m. in the morning

Figure 1: The knight on the gold trail.

In this assignment, you are expected to implement a grid-based shortest path finder algorithm to help the knight collect gold coins placed on a map. The project employs the **StdDraw graphics library** to dynamically visualize the map environment, tiles, and computed paths. The environment consists of three types of tiles representing different terrains: **grass tiles**, **sand tiles**, and **impassable tiles**. Your task is to read the given input files, compute the shortest path between the knight and the gold coins in a **specific order**, and visualize the navigation using the StdDraw library. You can watch this video to get a better understanding.

Tile Costs and Knight Movement

Your implementation should use a path-finding algorithm to find the shortest path from a source tile (knight's location) to a series of objectives (gold coins) **sequentially**. Travel cost between different tiles varies depending on their terrain type. Each type of terrain has an associated cost range, and the actual travel cost between tiles is assigned within these ranges. The travel costs for each terrain type are as follows:

- `Type 0:Grass` 1–5 units, low cost
- `Type 1:Sand` 8–10 units, mid-cost
- `Type 2:Obstacle` infinite, impassable

These values represent the cost of traveling *between* tiles. For example, if the knight travels from a grass tile to another grass tile, the cost of travel will vary between 1 and 5 units. If it travels between a grass tile and a sand tile, the cost of travel will vary between 8 and 10 units.

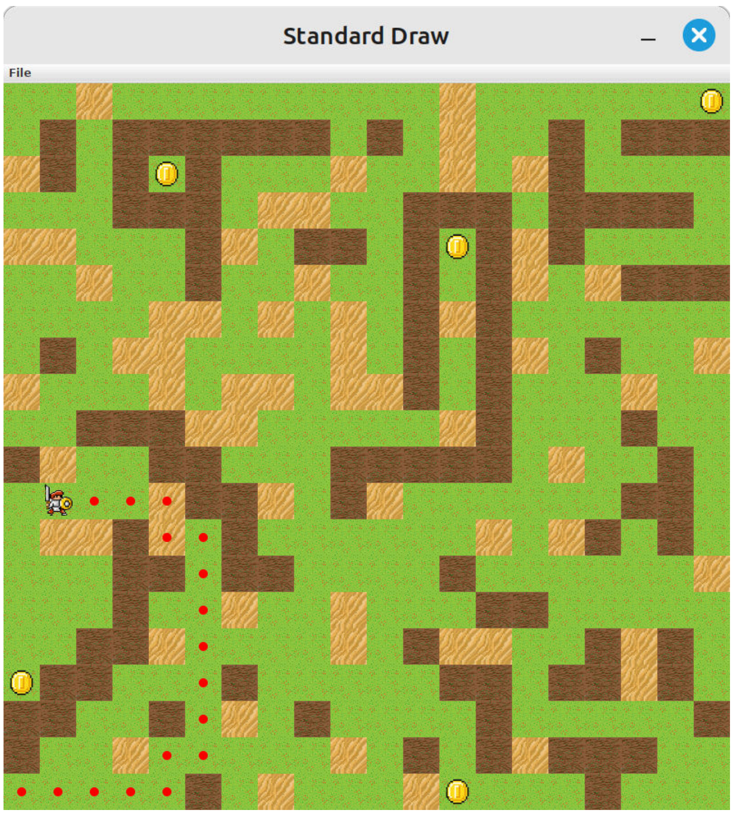

Figure 2: An example 20x20 Map of the Game.

The shortest path is computed by minimizing the total cost from the knight's current position to each gold coin. Once the shortest path is found, the path is visually represented by dots to show each step on the path using the StdDraw library, and the knight moves along the path until reaching the position of the gold coin. The visualization of the path can be seen in Figure 2, where the green tiles represent grass, the yellow tiles represent sand, and the brown tiles represent impassable obstacles. The knight is restricted to move in four directions: up, down, left, and right.

Input Files

Your program should take three input files as **command-line arguments**:

- `mapData.txt`: The first line of the file specifies the dimensions of the map in the format: # of columns, # of rows. The following lines represent tile coordinates and their terrain type in the format: `x y type`, where `x` represents the column number, `y` represents the row number, and `type` refers to its type. Types of the tiles are 0 for grass, 1 for sand, or 2 for impassable. The coordinate system assumes the coordinates of the top-left corner are (0, 0).
- `travelCosts.txt`: Specifies cost of travel between adjacent tiles in the format: `x1 y1 x2 y2 cost` where `x` and `y`'s represent the column and row number of the tiles, respectively. Impassable tiles have no defined cost, thus only the travel cost between passable tiles are given.
- `objectives.txt`: Specifies the coordinates of the starting point and the objectives. The first line contains the starting coordinates, followed by one or more lines with the objective coordinates, all in the format `x y`. Your program should find the shortest path from the starting position to the first objective. Once an objective is reached, it becomes the new starting point for the next objective. If an objective is unreachable, it should be skipped, and the current starting position should remain unchanged.

Output File

Your program should produce an output file named `output.txt` recording the shortest path. For example, if the coordinates of the starting position are `(0.0)` and the first objective is located at `(2,1)`, your output file should look like in the Codebox 1 below. Each line begins with the step number between the knight and the current objective, and the cost will be updated to reflect the total cost accumulated so far. Example output files covering different scenarios will be provided. The `output.txt` file should be placed in the `out` directory.

Code 1 : Example Output text

```
Starting position: (0, 0)
Step Count: 1, move to (1, 0). Total Cost: 1.97.
Step Count: 2, move to (1, 1). Total Cost: 4.53.
Step Count: 3, move to (2, 1). Total Cost: 12.55.
Objective 1 reached!
Starting position: (2, 1)
...
```

If an objective is impossible to reach, your program should not attempt to find a path. Instead, your program should print `"Objective n cannot be reached!"` and try to find a path to the next objective. An example output, assuming objective 2 is unreachable, is provided in Codebox 2 below. Moreover, the last line of the `output.txt` file should display the total steps and total cost of the entire journey in the format: `Total Step:#, Total Cost:#`

Code 2 : Example Output text

```
Starting position: (0, 0)
Step Count: 1, move to (1, 0). Total Cost: 1.97.
Step Count: 2, move to (1, 1). Total Cost: 4.53.
Step Count: 3, move to (2, 1). Total Cost: 12.55.
Objective 1 reached!
Objective 2 cannot be reached!
Starting position: (2, 1)
Step Count: 1, move to (2, 2). Total Cost: 8.52.
...
```

Implementation Details

Data Structures

Your implementation should utilize the `Tile` class for representing the tiles in the map. UML diagram of this class is provided in the Figure 3. You can implement additional methods, add new properties, or change the types of the properties. The UML diagram **only serves as a guide** for your implementation.

Tile	
-column: int	Column number of the Tile.
-row: int	Row number of the Tile.
-type: int	Type of the Tile.
-adjacentTiles: ArrayList<Tile>	Stores the neighbor `Tiles`
+Tile(column: int, row: int, type: int)	Constructs a Tile object.

Figure 3: UML Diagram of the `Tile` class.

It is also expected that you implement a class called `PathFinder` for finding the shortest path between the knight and **an objective**. You may define any necessary properties and methods for these classes and create additional classes as needed for this assignment.

Program Execution

Your program should accept `mapData.txt`, `travelCosts.txt` and `objectives.txt` as command line arguments. Different scenarios will be tested for your program, thus make sure that your program properly accepts command-line arguments and that your output files follow the **exact format** described earlier. You are also expected to include a `draw` flag for visualizing the path. If this flag is not used, only the `output.txt` file should be produced, and no animation should be displayed. Make sure that your program can be run **with and without using the `draw` flag** since the length and arguments in `args` will change. Regardless of whether the flag is used, your program should produce `output.txt`.

Your program must be compiled using the command provided below. The below command compiles all the Java source files under `code` directory and moves all resulting `.class` files to the `out` directory. Do not include `stdlib.jar`, any `.txt` files, or `out` directory in your submission, as we will use different files for testing.

```
javac -d out -cp "localPath/stdlib.jar" code/*.java
```

The second command runs the `Main` file in the `out` directory and draws the path using the `draw` flag.

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

On Windows machines, use backslashes (\) instead of slashes (/), and use ; instead of : in the classpath. Instead of `localPath/stdlib.jar`, use the actual local path on your machine—we will do the same with our local path during testing.

Bonus [10 points]

In the assignment, the order of the objectives (gold coins) is given with the `objectives.txt` so that your program finds the shortest path from the player's starting position to each gold coin in the given order. However, as a bonus, we challenge you to make your program collect all the gold coins in the most efficient way possible without any given order. Despite there are methods to solve this problem by approximating the shortest route, you are expected to find the exact shortest route that visits every objective beginning from the starting point. The journey of the knight should end when he returns to the tile where he started.

When we run your program in our machines, it should find the shortest route within three seconds. Your program will be tested with at least 15 objectives (gold coins) for this task. In addition to the aforementioned classes, you are expected to create a class named `ShortestRoute` to compute the shortest route that begins at the source tile and visits all the objectives. The output file generated for this task should be named `bonus.txt` and placed in the `out` directory.

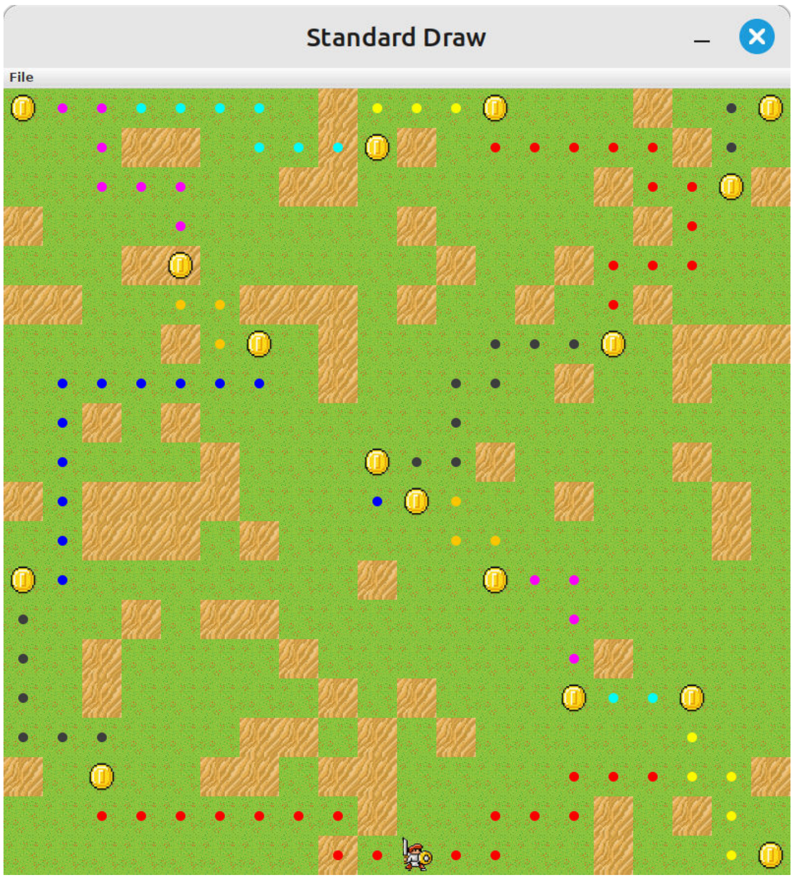

Figure 4: Knight's shortest route to collect all coins.

Report Content

Your report should be concise and consist of the following sections:

- **Introduction:** Briefly explain the project and its components.
- **Class Diagrams:** Explain the classes you implemented with their UML diagrams.
- **Algorithm:** Explain the algorithm you implemented and how you used the classes you have defined.
- **Bonus:** If you implement the bonus part, explain the algorithm of your implementation.

Your report should **properly reference all figures, tables, and equations**. Figures and tables should be numbered and cited appropriately (as shown in Figure 1, etc.). Each figure must have a caption placed below it, clearly explaining its content. Each table must have a caption positioned above it. Lastly, all equations should be centered and numbered for reference.

Notes

- You **must** implement in Object-oriented way.
- You **must** use the **StdDraw graphics library**.

Submission Files

1. Java source codes (**.java files**). Submission details are explained in greater detail in the Submission Guide section.
2. Report (**.pdf format**) explaining your implementation. Your report should not exceed 8 pages; if you implement the bonus part, then 10 pages maximum.

Submission Guide

Submission Files

Submit a single compressed (`.zip`) file, named as `NameSurname.zip`, to Moodle. It should contain all source code files (under the `\code` directory), the report (in **PDF** format, under the `\report` directory), and all other files if needed (under the `\misc` directory). Name the main Java file as `Main.java`, if you implement the bonus part, then create a second file as `Bonus.java`. Name your report as `NameSurname.pdf`. Do not use Turkish characters in filenames, code, or comments. In each `.java` file, add your name and student number as a comment at the top.

Late Submission Policy

The maximum late submission duration is two days. Late submissions will be graded at 50% of the original grade.

Mandatory Submission

Submission of assignments is **mandatory**. If you do not submit an assignment, you **fail** the course.

Plagiarism

This leads to a grade of F, and YÖK regulations will be applied.