CmpE 160 Assignment 3

Gold Trail: The Knight’s Path

Student Name: Devin Isler

Student ID: 2023400063

Date: May 9, 2025

Introduction

This report details the implementation of the "Gold Trail: The Knight's Path" project. The project implements a grid-based shortest path finding algorithm to assist a knight character in navigating through a terrain map to collect gold coins in the most efficient manner possible.

The core challenge of this project involves constructing an efficient pathfinding algorithm that can calculate the optimal route between the knight's current position and a series of objective points (gold coins) while considering the varying travel costs associated with different terrain types. The implementation utilizes object-oriented programming principles to create a robust solution that can handle complex map environments.

The project employs the StdDraw graphics library to visualize the map environment, which consists of three distinct terrain types: grass tiles (low cost), sand tiles (mid-cost), and impassable obstacle tiles. The knight's movement is restricted to four cardinal directions (up, down, left, right), and the goal is to minimize the total travel cost while collecting all gold coins in a predefined sequence.

This report describes the classes implemented, the path-finding algorithm utilized, and the approach taken to handle different scenarios, including unreachable objectives. The implementation takes input from the terminal, reads map data, travel costs, and objectives from input files, computes the shortest path to each objective sequentially, and produces a detailed output file documenting the knight's journey.

Class Implementations

- **Tile Class:**

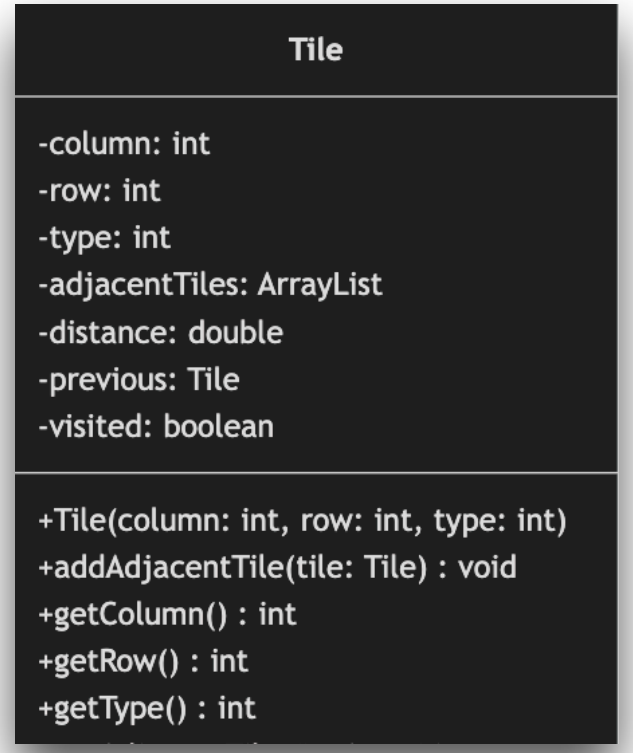

Figure 1: UML diagram of Tile Class

The *Tile* class (Figure 1) represents an individual tile on the game map, encapsulating its position, terrain type, and pathfinding properties. It is fundamental to map representation and navigation. Each tile stores its column and row coordinates, type (0 for grass, 1 for sand, 2 for obstacle), and a list of adjacent tiles for movement. Additional attributes, such as distance, previous tile, and visited status, support *Dijkstra's* algorithm in pathfinding. Key methods include constructors for initialization, getters for accessing properties, and methods to manage adjacency and pathfinding states.

- **PathFinder Class:**

Figure 2: UML diagram of PathFinder Class

The *PathFinder* class (Figure 2) is responsible for computing the shortest path between two tiles using *Dijkstra's* algorithm. It operates on the map's tile array, dimensions, and travel costs provided as input. The class initializes with the map and cost data, and its primary method, *findShortestPath*, calculates the optimal path from a source tile to a target tile, respecting terrain costs and avoiding obstacles. A private helper method, *getCost*, retrieves the travel cost between adjacent tiles. This class is used by both *Main* and *Bonus* to navigate the knight to objectives.

- **Main Class**

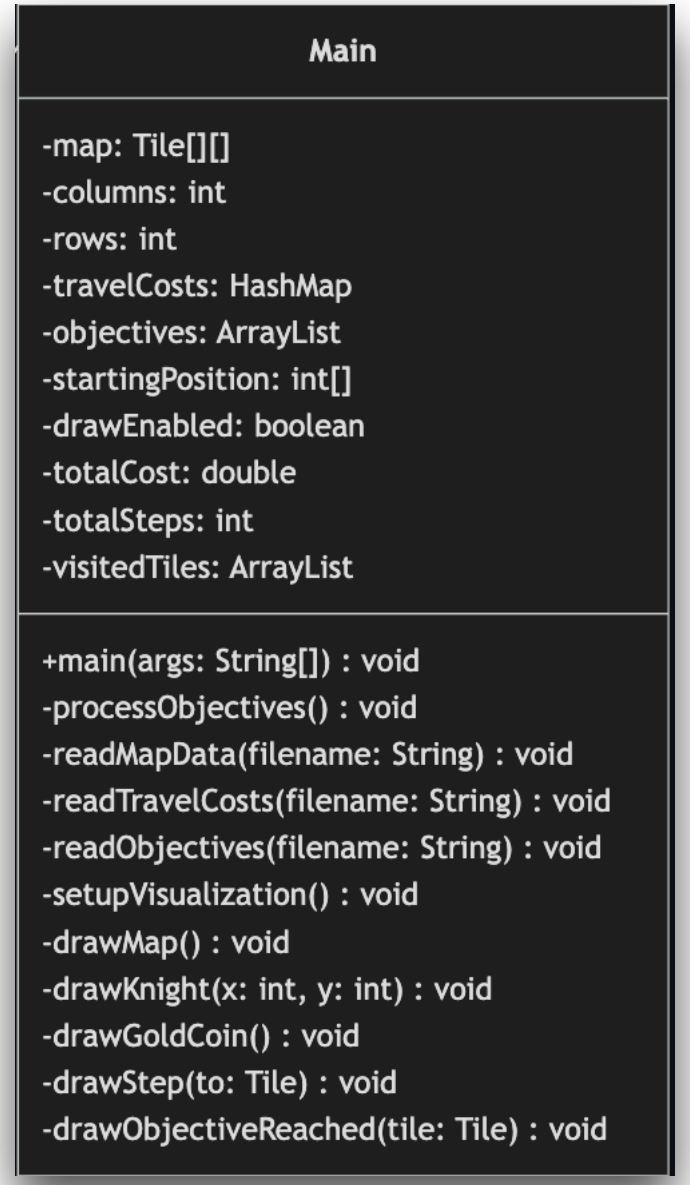

Figure 3: UML Diagram of Main Class

The Main class serves as the central hub for the standard assignment, orchestrating the knight's quest to collect gold coins. It gets input from terminal, processes input files (mapData.txt, travelCosts.txt, objectives.txt) to initialize the game map and coordinates the *PathFinder* class to compute shortest paths to each objective in sequence. With an optional *-draw* flag, it employs *StdDraw* to visualize the map, knight, and paths. The class generates detailed path information, including steps and cumulative costs, in *output.txt*.

- **ShortestRoute Class:**

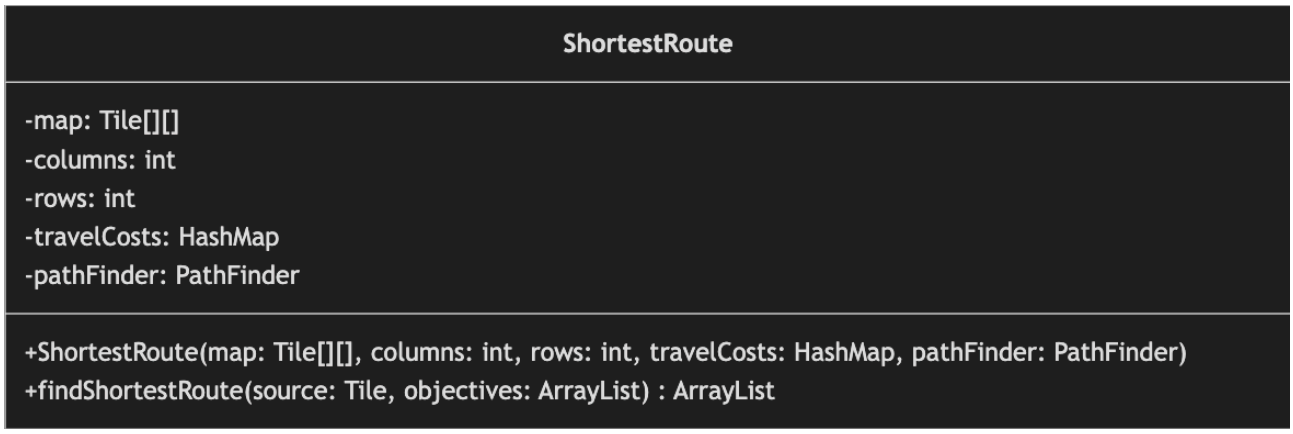

Figure 4: UML Diagram of ShortestRoute Class

The *ShortestRoute* class (Figure 4) addresses the bonus requirement by solving the T*raveling Salesman Problem* to find the shortest route that starts at a source tile, visits all objective tiles, and returns to the source. It integrates with the *PathFinder* class to compute pairwise shortest paths and uses dynamic programming to determine the optimal visitation order. The class stores map data and a *PathFinder* instance, with its main method, *findShortestRoute*, returning the ordered list of tiles. This class is exclusively used by the *Bonus* class to optimize the knight's journey.

- **Bonus Class:**

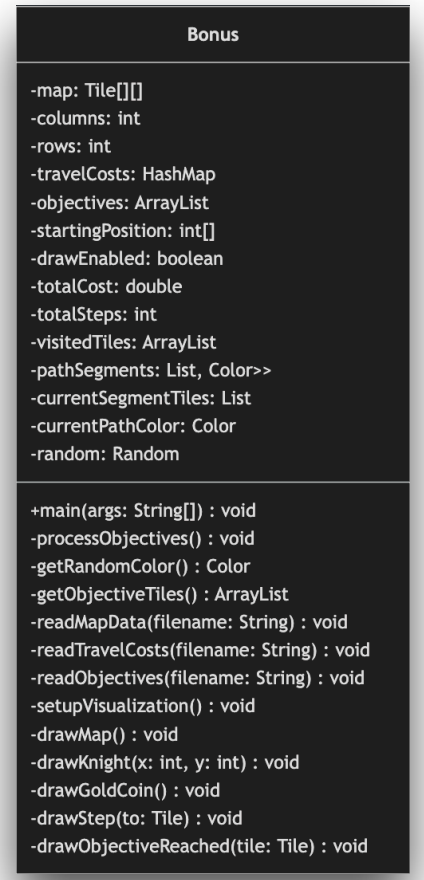

Figure 5: UML Diagram for Bonus Class

The *Bonus* class implements the bonus task, optimizing the knight's journey to collect all gold coins and return to the starting point. Like *Main,* it accepts input from terminal, processes input files (mapData.txt, travelCosts.txt, objectives.txt) and employs the *ShortestRoute* class to compute the most efficient route using the *Traveling Salesman Problem* solution. With the *-draw* flag, it enhances visualization by displaying colored path segments via *StdDraw*. The class records the optimized path, steps, and costs in bonus.txt. Key methods manage input parsing, route processing, color generation, and dynamic visualization.

```
        // Process the queue
        while (!queue.isEmpty()) {
            Tile current = queue.poll();

            // If we've reached the target, we're done
            if (current.equals(target)) {
                break;
            }

            // Skip if already visited
            if (current.isVisited()) {
                continue;
            }

            current.setVisited(true);

            // Process each adjacent tile
            ArrayList<Tile> neighbors = current.getAdjacentTiles();
            for (int i = 0; i < neighbors.size(); i++) {
                Tile neighbor = neighbors.get(i);

                // Skip obstacles (type 2)
                if (neighbor.getType() == 2) {
                    continue;
                }

                // Get the travel cost between current and neighbor
                double cost = getCost(current, neighbor);
                double newDistance = current.getDistance() + cost;

                // Update distance if we've found a shorter path
                if (newDistance < neighbor.getDistance()) {
                    neighbor.setDistance(newDistance);
                    neighbor.setPrevious(current);

                    // Add to queue for processing
                    queue.add(neighbor);
```

Figure 6: Core Loop of Dijkstra's Algorithm in PathFinder

Algorithm For the Normal Part:

The standard assignment enables a knight to collect gold coins sequentially on a grid-based map, minimizing travel costs. The *Main* class orchestrates the process by parsing three input files: *mapData.txt* (specifying map dimensions and tile types), *travelCosts.txt* (defining movement costs between adjacent tiles), and *objectives.txt* (listing the starting position and coin coordinates). The *Tile* class models each map tile, storing its column, row, terrain type (0: grass, 1: sand, 2: obstacle), and adjacent tiles, forming a graph for pathfinding. The *Main* class initializes a *2D Tile* array, populating it with tile data and linking adjacent tiles for navigation.

The *PathFinder* class implements *Dijkstra's algorithm* to compute the shortest path between the knight's current position and each objective. *Dijkstra's algorithm* proceeds as follows:

1. **Initialization:** Set the distance of the source tile (the knight's current position) to 0 and all other tiles to infinity. Mark all tiles as unvisited and insert the source tile into a priority queue.

2. **Tile Selection:** Repeatedly extract the unvisited tile with the smallest distance from the priority queue. Once a tile is extracted, it is considered visited and will not be processed again.

3. **Neighbor Processing:** For each unvisited neighboring tile (excluding obstacles of type 2), calculate a new tentative distance as the current tile's distance plus the travel cost. If this new distance is less than the neighbor's recorded distance, update the neighbor's distance and store the current tile as its predecessor. Add the neighbor to the priority queue if not already present.

4. **Termination:** This process continues until the target tile is reached or the priority queue is empty (indicating no path exists).

5. **Path Reconstruction:** If a path to the target was found, reconstruct it by following the predecessor pointers from the target tile back to the source.

The Figure 6 above illustrates the core loop of *Dijkstra's* algorithm, managing the priority queue and distance updates.

The *Main* class iteratively invokes *PathFinder* for each objective, starting from the initial position and updating the knight's position to the reached objective. Each step's coordinates and cumulative cost are written to *output.txt*. If an objective is unreachable (e.g., surrounded by obstacles), it is is declared as not reachable and proceeds to the next one. With the *-draw* flag, *Main* uses *StdDraw* to render the map (grass, sand, obstacles), knight, coins, and path dots, updating the display per step as shown in Figure 7. The algorithm concludes by appending total steps and costs to *output.txt*, ensuring efficient pathfinding and accurate visualization.

```
263     /**
264      * Sets up the StdDraw visualization.
265      */
266     private static void setupVisualization() {  1 usage
267         StdDraw.setCanvasSize(30*rows, 30*columns);
268         StdDraw.setXscale(-0.5, columns - 0.5);
269         StdDraw.setYscale(-0.5, rows - 0.5);
270         StdDraw.enableDoubleBuffering();
271
272         // Draw the map
273         drawMap();
274
275         // Draw objectives
276         drawGoldCoin();
277
278         // Draw knight at starting position
279         drawKnight(startingPosition[0], startingPosition[1]);
280
281         StdDraw.show();
282     }
```

Figure 7: Visualization

Algorithm For the Bonus Part:

The bonus task optimizes the knight's journey to visit all gold coins and return to the starting point, solving the *Traveling Salesman Problem (TSP)*. The *Bonus* class initiates the process by reading input files: *mapData.txt* (map dimensions and tile types), *travelCosts.txt* (movement costs), and *objectives.txt* (starting position and coin locations). The *Tile* class represents tiles with coordinates, terrain types (grass, sand, obstacle), and adjacency, forming a navigable graph. The *ShortestRoute* class computes the optimal route using dynamic programming, leveraging the *PathFinder* class for pairwise shortest paths.

The *TSP* solution proceeds as follows:

1. **Pairwise Path Computation**: As can be seen in Figure 8, *ShortestRoute* uses *PathFinder's Dijkstra's* algorithm to calculate shortest path costs between the starting tile and all objectives, and among all objective pairs, storing costs in a matrix.

2. **Dynamic Programming**: Define a state table *dp[mask][last]*, where mask is a bitmask representing visited tiles, and last is the last visited tile. *dp[mask][last]* stores the minimum cost to visit all tiles in mask ending at last. Set *dp[1][0]* to *0* (source tile visited). For each subset mask:
   - Iterate over last (tiles in *mask*).
   - For each unvisited tile next, compute the new cost as the sum of *dp[mask][last]* and *costs[last][next]*.
   - Update *dp[newMask][next]* if the new cost is lower, where *newMask* includes next.
   - Store the previous tile in *parent[newMask][next]* for route reconstruction.

   Figure 9 below illustrates the structure of this dynamic programming approach, including the bitmask states, cost updates, and parent tracking for route reconstruction.

3. **Route Construction**: Identify the minimum cost to visit all objectives and return to the source, then reconstruct the route by backtracking through parent pointers.

4. **Path Execution**: *Bonus* follows the optimal route, using *PathFinder* to compute detailed paths between consecutive tiles, accumulating steps and costs.

The *Bonus* class writes each step's coordinates and cumulative cost to *bonus.txt*, noting reached objectives. If a path is unreachable, it logs an error. With the *-draw* flag, *Bonus* visualizes the map, knight, coins, and paths using *StdDraw* (Figure 7), assigning random colors to path segments for clarity. The algorithm concludes with total steps and costs, ensuring an efficient, visually distinct solution within three seconds for 15+ objectives.

```
55        // Precompute shortest path costs between all pairs of tiles (source + objectives)
56        Tile[] nodes = new Tile[n + 1];
57        nodes[0] = source;
58        for (int i = 0; i < n; i++) {
59            nodes[i + 1] = objectives.get(i);
60        }
61
62        double[][] costs = new double[n + 1][n + 1];
63        for (int i = 0; i <= n; i++) {
64            for (int j = 0; j <= n; j++) {
65                if (i == j) {
66                    costs[i][j] = 0;
67                } else {
68                    ArrayList<Tile> path = pathFinder.findShortestPath(nodes[i], nodes[j]);
69                    costs[i][j] = path != null ? nodes[j].getDistance() : Double.POSITIVE_INFINITY;
70                }
71            }
72        }
```

Figure 8: Pairwise Shortest Path Computation in ShortestRoute

```java
        // dp[mask][last] = min cost to visit all nodes in mask ending at last
        double[][] dp = new double[1 << (n + 1)][n + 1];
        int[][] parent = new int[1 << (n + 1)][n + 1];
        for (double[] row : dp) {
            Arrays.fill(row, Double.POSITIVE_INFINITY);
        }

        // Initialize: start at source (node 0)
        dp[1][0] = 0;

        // Iterate over all subsets
        for (int mask = 1; mask < (1 << (n + 1)); mask++) {
            for (int last = 0; last <= n; last++) {
                if (dp[mask][last] == Double.POSITIVE_INFINITY) continue;
                // Try adding each unvisited node
                for (int next = 0; next <= n; next++) {
                    if ((mask & (1 << next)) == 0 && costs[last][next] != Double.POSITIVE_INFINITY) {
                        int newMask = mask | (1 << next);
                        double newCost = dp[mask][last] + costs[last][next];
                        if (newCost < dp[newMask][next]) {
                            dp[newMask][next] = newCost;
                            parent[newMask][next] = last;
                        }
                    }
                }
            }
        }
```

Figure 9: Dynamic Programming for TSP in ShortestRoute