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

Report

Ahmet Mete Atay
2023400240

Abstract

This report outlines the implementation of a grid-based pathfinding system for the "Gold Trail: The Knight's Path" assignment. The project involves a knight collecting gold coins on a map with varied terrain. The primary task uses a shortest path algorithm for sequential objective collection, while an optional bonus task seeks the most efficient tour for all coins. Visualization is achieved using the StdDraw library. This document covers class design and algorithmic approaches.

1. Introduction

The "Gold Trail" project requires guiding a knight on a 2D grid map (grass, sand, impassable tiles) to collect gold coins. The main task involves finding the shortest path to a sequence of objectives, minimizing travel costs. Input files (mapData.txt, travelCosts.txt, objectives.txt) define the environment and goals. The program visualizes movement via StdDraw and logs details to output.txt.

The optional bonus involves solving the Traveling Salesperson Problem (TSP): finding the optimal route to visit all coins and return to the start, outputting to bonus.txt. This report details the system's object-oriented design and algorithms.

2. Class Diagrams

The system uses several key classes: Tile, PathFinder (main assignment), ArrayPair (utility), ShortestRouteSolver (bonus), and DijkstraUtil (bonus helper).

2.1. Tile Class

Represents a map square with terrain type, coordinates, and neighbors.

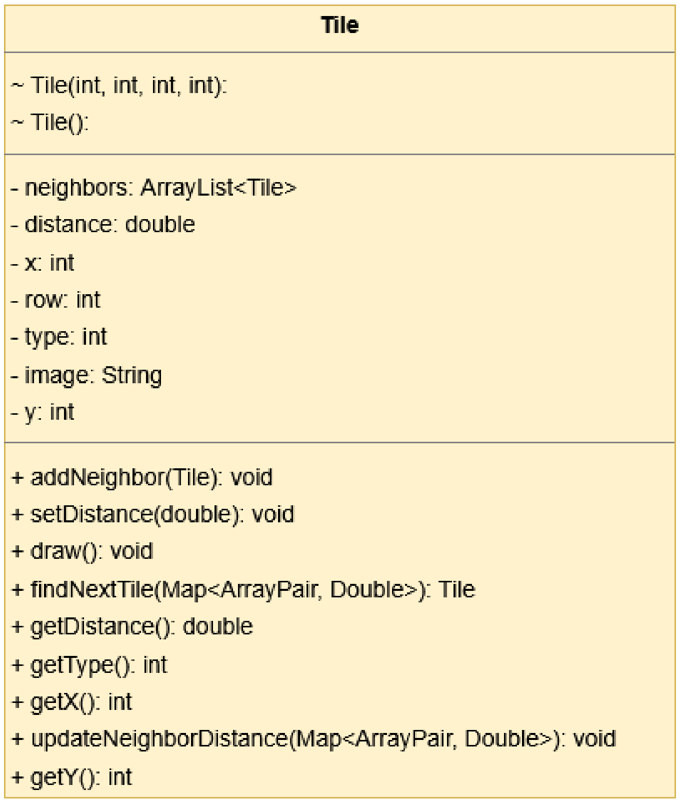

Tile
~ Tile(int, int, int, int):
~ Tile():
- neighbors: ArrayList<Tile>
- distance: double
- x: int
- row: int
- type: int
- image: String
- y: int
+ addNeighbor(Tile): void
+ setDistance(double): void
+ draw(): void
+ findNextTile(Map<ArrayPair, Double>): Tile
+ getDistance(): double
+ getType(): int
+ getX(): int
+ updateNeighborDistance(Map<ArrayPair, Double>): void
+ getY(): int

Figure 1:UML diagram for Tile Class

2.2. PathFinder Class

Calculates the shortest path to a single objective.

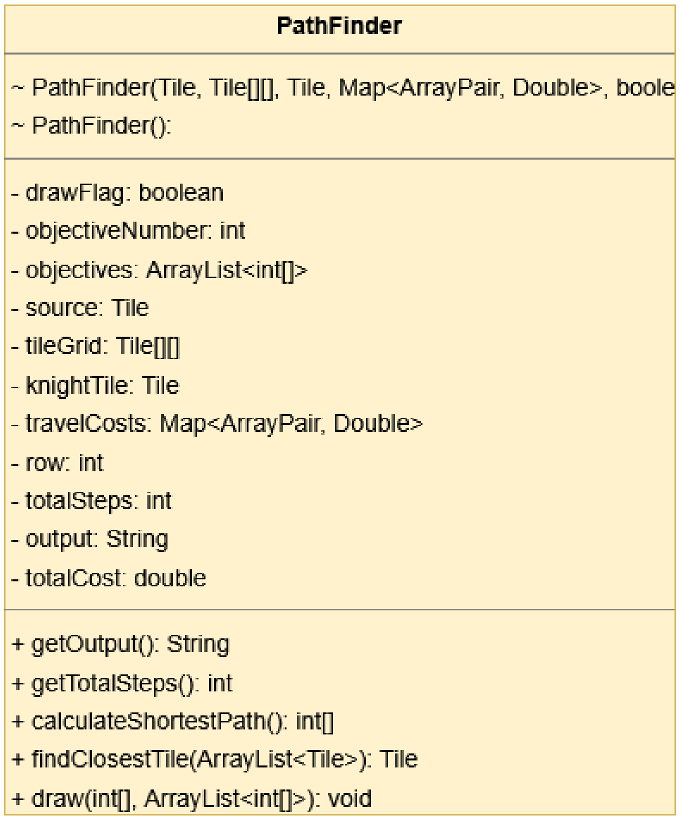

PathFinder
~ PathFinder(Tile, Tile[][], Tile, Map<ArrayPair, Double>, boole
~ PathFinder():
- drawFlag: boolean
- objectiveNumber: int
- objectives: ArrayList<int[]>
- source: Tile
- tileGrid: Tile[][]
- knightTile: Tile
- travelCosts: Map<ArrayPair, Double>
- row: int
- totalSteps: int
- output: String
- totalCost: double
+ getOutput(): String
+ getTotalSteps(): int
+ calculateShortestPath(): int[]
+ findClosestTile(ArrayList<Tile>): Tile
+ draw(int[], ArrayList<int[]>): void

Figure 2:UML diagram for PathFinder Class

2.3. ArrayPair Class

Utility for HashMap keys representing tile pairs for travel costs.

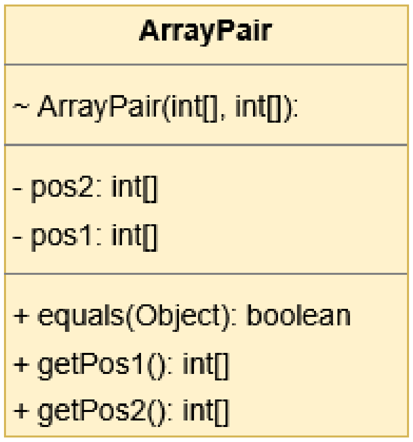


Figure 3:UML diagram for ArrayPair Class

2.4. DijkstraUtil Class (Bonus Assignment)

Provides a static Dijkstra's algorithm implementation.

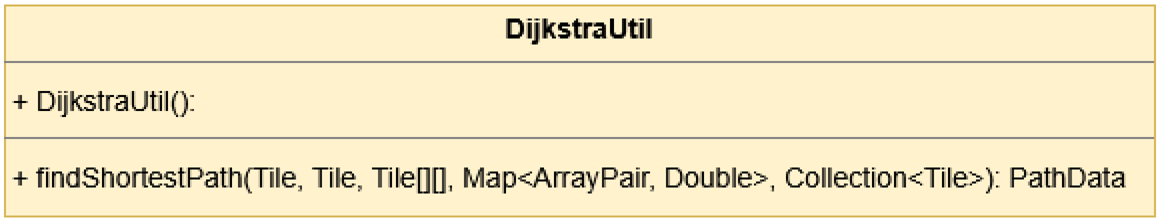
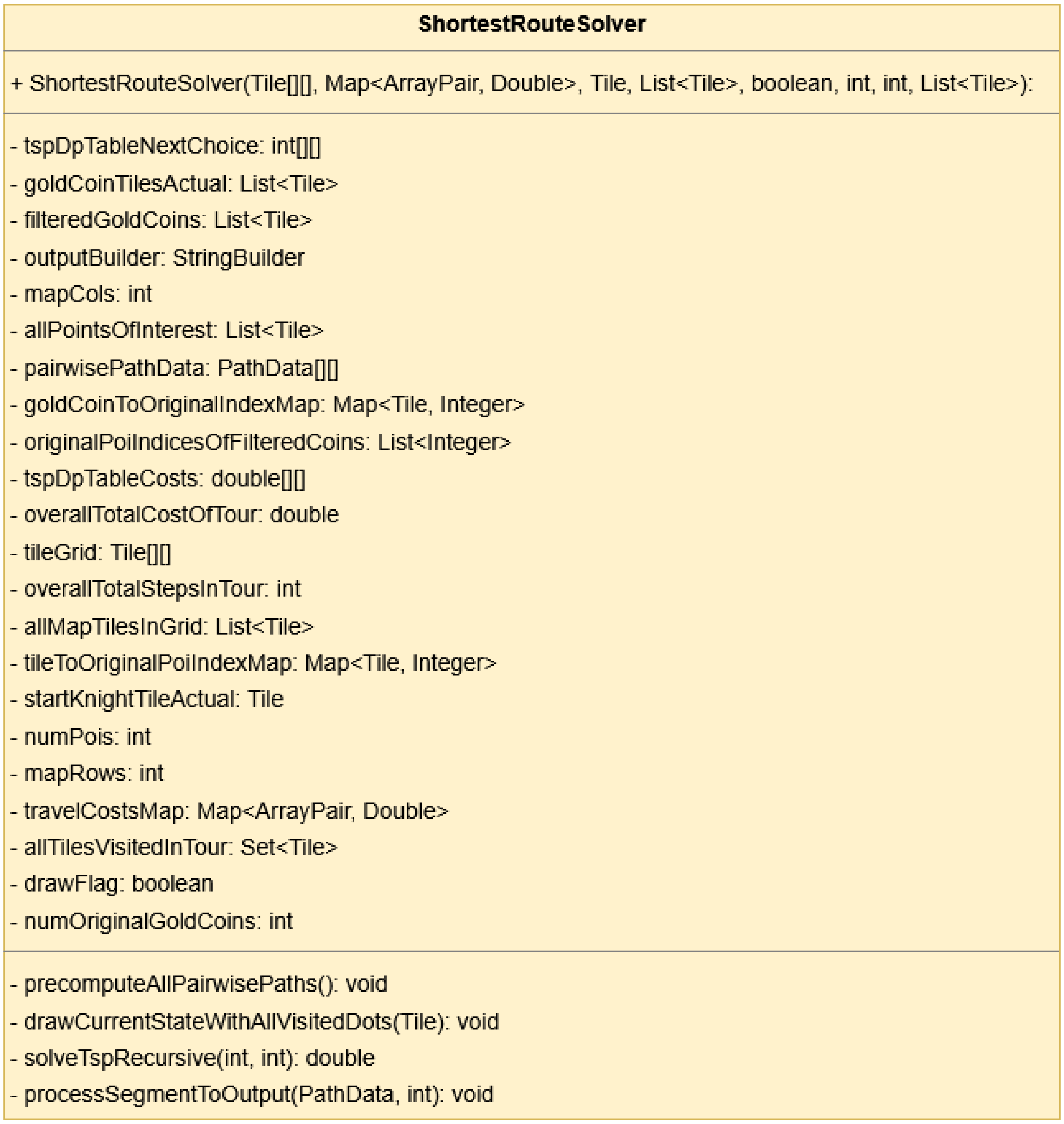

DijkstraUtil
+ DijkstraUtil():
+ findShortestPath(Tile, Tile, Tile[][], Map<ArrayPair, Double>, Collection<Tile>): PathData

Figure 4:UML diagram for DijkstraUtil Class

2.5. ShortestRouteSolver Class (Bonus Assignment)

Solves the TSP for the bonus task.



ShortestRouteSolver

+ ShortestRouteSolver(Tile[][], Map<ArrayPair, Double>, Tile, List<Tile>, boolean, int, int, List<Tile>):

- tspDpTableNextChoice: int[][]
- goldCoinTilesActual: List<Tile>
- filteredGoldCoins: List<Tile>
- outputBuilder: StringBuilder
- mapCols: int
- allPointsOfInterest: List<Tile>
- pairwisePathData: PathData[][]
- goldCoinToOriginalIndexMap: Map<Tile, Integer>
- originalPoiIndicesOfFilteredCoins: List<Integer>
- tspDpTableCosts: double[][]
- overallTotalCostOfTour: double
- tileGrid: Tile[][]
- overallTotalStepsInTour: int
- allMapTilesInGrid: List<Tile>
- tileToOriginalPoiIndexMap: Map<Tile, Integer>
- startKnightTileActual: Tile
- numPois: int
- mapRows: int
- travelCostsMap: Map<ArrayPair, Double>
- allTilesVisitedInTour: Set<Tile>
- drawFlag: boolean
- numOriginalGoldCoins: int

- precomputeAllPairwisePaths(): void
- drawCurrentStateWithAllVisitedDots(Tile): void
- solveTspRecursive(int, int): double
- processSegmentToOutput(PathData, int): void

Figure 5:UML diagram for ShortestRouteSolver Class

3. Algorithm

3.1. Main Assignment: Sequential Path Finding

The knight visits objectives in a given order.

- Input & Setup: Main.java reads map, travel costs, and objectives. Tiles are stored in tileGrid, costs in a HashMap<ArrayPair, Double>.
- Pathfinding (PathFinder.calculateShortestPath): A Dijkstra-like algorithm finds the shortest path from the current objective back to the knight.
    1. Initialize objective's distance to 0, others to infinity.
    2. Iteratively select the unsettled tile closest to the objective, update its neighbors' distances, and mark it settled.
    3. If the knight's tile is settled, reconstruct the path from the knight to the objective by following the gradient of decreasing distances. If the knight's tile cannot be settled, the objective is unreachable.
- Execution & Output: This process repeats for each objective. Steps and costs are logged.
    - Visualization: If -draw is used, StdDraw shows the map, knight, coins, and path.

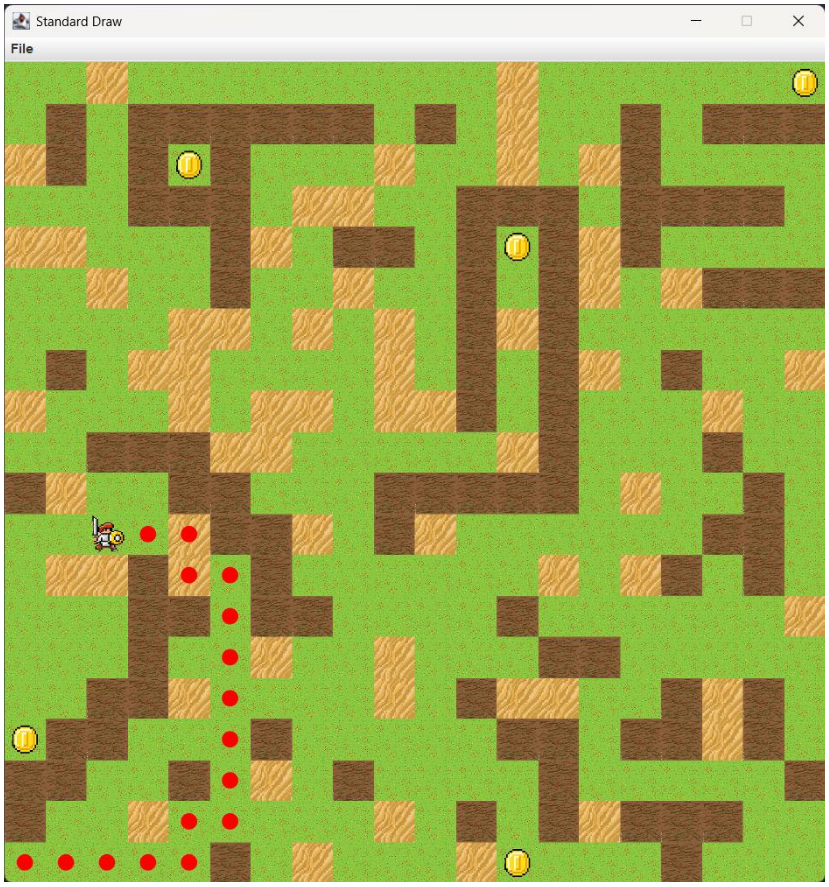


Figure 6: Screenshot showing map, knight, coins, and path dots for a segment.

3.2. Bonus Assignment: Optimal Tour (TSP)

Finds the shortest tour visiting all coins and returning to the start.

- Step 1: All-Pairs Shortest Paths: ShortestRouteSolver uses DijkstraUtil.findShortestPath() to compute shortest path costs and tile sequences between all Points of Interest (POIs: start tile and all original gold coins), stored in pairwisePathData.

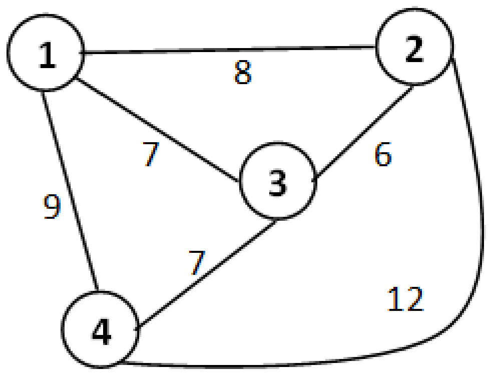


Figure 7: Conceptual POI graph with pairwise path costs

- Step 2: Filter Unreachable Coins: Coins not reachable from the start (and vice-versa) are excluded from the TSP.
- Step 3: TSP with Dynamic Programming: The Held-Karp style algorithm finds the optimal visit order for filteredGoldCoins.
    - State dp[i][mask]: Min cost to visit coins in mask, ending at filtered coin i, then returning to the actual start.
    - Solved recursively with memoization (tspDpTableCosts, tspDpTableNextChoice).
- Step 4: Path Reconstruction & Output: The best first coin from the actual start is determined. The optimal tour is reconstructed. The order of visiting coins in this tour is reversed before generating the step-by-step output to bonus.txt, using segments from pairwisePathData.
- Visualization: If enabled, StdDraw shows the knight traversing the complete optimal tour.

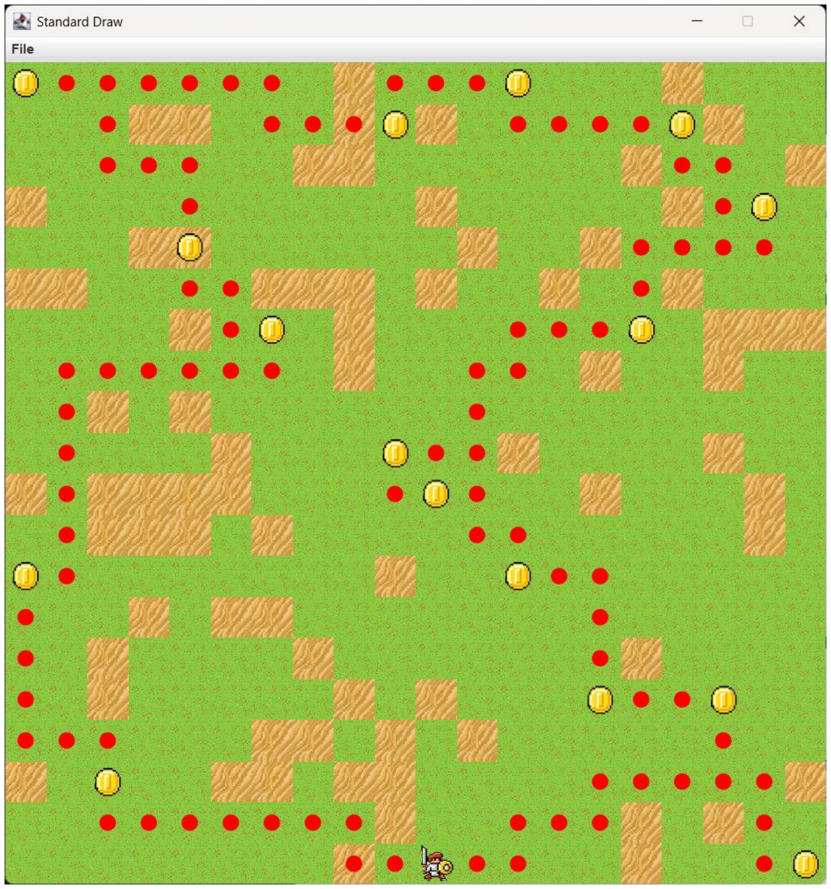


Figure 8: Screenshot showing map, knight, coins, and the complete optimal tour path dots.

4. Conclusion

The project successfully implements a pathfinding system for the knight. The main assignment uses a sequential Dijkstra-based approach, while the bonus tackles the TSP with dynamic programming and precomputed pairwise paths. The object-oriented design facilitates modularity, and StdDraw provides effective visualization. Both parts meet requirements by producing detailed output logs.