Gold Trail: The Knight's Path - Project Report

Introduction

The **Gold Trail** project is a comprehensive pathfinding application developed in Java, aimed at navigating a knight through a grid-based environment filled with different terrain types to collect gold coins. The knight starts from a designated tile and visits each objective tile by finding the most efficient (least cost) route. The environment includes three kinds of terrain:

- **Grass (Type 0)**: Low-cost tile with a travel cost between 1 and 5.
- **Sand (Type 1)**: Medium-cost tile with a travel cost between 8 and 10.
- **Impassable (Type 2)**: These tiles cannot be traversed.

The knight can only move in the four cardinal directions: up, down, left, and right. Visualization of movement is provided using the **StdDraw** library. The project not only aims to provide correct functionality but also adheres to object-oriented principles, modular design, and clean code structure.

In addition to implementing a sequential pathfinding solution using **Dijkstra's algorithm**, the project also includes a **bonus component**: finding the shortest round-trip path that visits all objectives exactly once and returns to the starting point — a variation of the well-known **Traveling Salesman Problem (TSP)**.

Class Diagrams

Tile Class

```
+----------------+
|    Tile      |  // Represents each individual tile on the map
+----------------+
| - column: int  |  // X-coordinate of the tile
| - row: int     |  // Y-coordinate of the tile
| - type: int    |  // 0 = grass, 1 = sand, 2 = impassable
| - neighbors: Map<Tile, Double> | // Adjacent tiles and edge costs
+----------------+
| +Tile(col, row, type) |
```

```
| +equals(Object): boolean |
| +hashCode(): int       |
+----------------+
```

The Tile class models an individual tile in the map, storing terrain type and connections to neighboring tiles with their respective movement costs.

MapHandler Class

```
+----------------------+
|    MapHandler       |  // Responsible for loading and storing map data
+----------------------+
| - tiles: Map<String, Tile> | // All tiles stored by key "x,y"
| - width: int         |  // Width of the map
| - height: int        |  // Height of the map
+----------------------+
| +loadMap(filePath)    |
| +loadTravelCosts(filePath) |
| +loadObjectives(filePath): List<int[]> |
| +key(x, y): String    |
+----------------------+
```

MapHandler handles file input operations. It reads and processes the map structure, inter-tile movement costs, and objectives.

PathFinder Class

```
+-----------------------------+
|       PathFinder         | // Core logic for step-by-step pathfinding
+-----------------------------+
| - mapHandler: MapHandler     |
+-----------------------------+
| +PathFinder(mapHandler)      |
| +findPaths(objectives)       |
| +dijkstra(start, end): Result|
| -drawMap()                |
| -setupStdDraw()            |
+-----------------------------+
```

Responsible for sequential navigation between objectives. It uses Dijkstra's algorithm for each leg of the journey and visualizes the process.

Bonus Class

```
+-------------------------+
|       Bonus         | // Optimizes route across all objectives
+-------------------------+
| - mapHandler: MapHandler|
+-------------------------+
| +findShortestRoute(objectives) |
| -drawMap(...)         |
| -key(...)             |
+-------------------------+
```

Bonus calculates the most efficient overall path visiting all objectives and returning to the start, solving a variation of the TSP.

Algorithm: Dijkstra's Pathfinding in Depth

Dijkstra's algorithm is a single-source shortest path algorithm. It works for graphs with non-negative edge weights and is ideal for our terrain, where travel costs are always ≥ 0.

Conceptual Overview

- The graph is composed of **nodes** (tiles) and **edges** (movements between tiles).
- Each edge has a cost depending on terrain.
- The algorithm keeps track of the shortest known path to each tile using a **distance map** (Map<Tile, Double>).

Step-by-step Execution

1. **Initialization**:
    a. Set all distances to Infinity except the start tile which is set to 0.0.
    b. Add the start tile to a **priority queue**, sorted by current known distance.
2. **Relaxation Loop**:
    a. Extract the tile with the lowest current distance.
    b. For each neighbor, calculate tentative distance: alt = dist[u] + cost(u, v).
    c. If alt < dist[v], update dist[v] and record u as the previous tile.
    d. Reinsert the neighbor into the queue if needed.

3. **Path Construction**:
    a. Once the destination is reached, reconstruct the path using the prev map.
4. **Visualization**:
    a. Each tile visited is drawn using StdDraw.
    b. Movement is logged in output.txt.

```
while (!queue.isEmpty()) {
  Tile u = queue.poll();
  if (u.equals(end)) break;

  for (Map.Entry<Tile, Double> entry : u.neighbors.entrySet()) {
    Tile v = entry.getKey();
    double alt = dist.get(u) + entry.getValue();
    if (alt < dist.get(v)) {
      dist.put(v, alt);
      prev.put(v, u);
      queue.add(v);
    }
  }
}
```

The result is stored in a custom Result class that encapsulates the shortest path (List<Tile> path) and the cost to reach each tile (Map<Tile, Double> costs). This makes it easy to reuse the results, particularly in the bonus part.

Code Architecture and Execution Flow

Main.java

The main entry point reads input files and invokes the PathFinder or Bonus class depending on whether the -draw flag is used or if the bonus functionality is triggered. It uses command-line arguments to ensure flexibility and testability:

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

findPaths() in PathFinder

This method loops through all objectives and calls dijkstra() for each segment. It handles exceptions for unreachable objectives and continues traversal from the last successful node:

```
for (int i = 1; i < objectives.size(); i++) {
  Result result = dijkstra(currentObjective, target);
  if (result == null) {
    writer.println("Objective " + i + " cannot be reached!");
  } else {
    // Draw and animate path
  }
}
```

drawMap()

This helper method paints the entire map with terrain and objectives using StdDraw. Objective tiles are drawn as coins (coin.png) and visited tiles are highlighted with red dots.

Error Handling

- All file operations are enclosed in try-catch blocks.
- Unreachable objectives are gracefully handled and do not crash the program.

Output Files

- output.txt: Generated in standard mode.
- bonus.txt: Generated in bonus mode.

Both follow a structured logging format that matches the specification given in the assignment PDF.

Bonus: Greedy Approximation to TSP

The bonus part significantly increases complexity: rather than going from point A to B to C in order, we must find the best order for visiting all targets and return to the origin.

Why Not Brute Force?

Permutations of n objectives is n!. For 15 objectives, that's 1.3 x 10^12 permutations. This is infeasible to compute within 3 seconds.

Heuristic Solution: Nearest Neighbor + Preprocessing

Step 1: Precompute all pairwise shortest paths

- Using Dijkstra's algorithm for each objective pair.
- Store in a Map<String, Result> indexed by a unique key fromX,fromY->toX,toY.

Step 2: Greedy nearest-neighbor traversal

- Start at initial tile.
- While there are unvisited objectives:
  - Choose the closest remaining objective (using precomputed results).
  - Move the knight there.
  - Remove from the objective pool.

Step 3: Return to starting point

- Append the shortest return route to the tour.

Pseudocode Summary:

```
List<Integer> remaining = allObjectiveIndices;
while (!remaining.isEmpty()) {
  findNearestObjective();
  update current;
  mark as visited;
}
returnToStart();
```

Advantages of This Heuristic

- Fast and scalable (executes in under 3 seconds)
- Deterministic and predictable
- Produces reasonably optimal paths (though not mathematically guaranteed)

Sample Logging Output

Step Count: 12, move to (3, 4). Total Cost: 76.50.
Objective 4 reached at (3, 4)!
...
Step Count: 53, move to (0, 0). Total Cost: 138.70.
Total Step: 52, Total Cost: 138.70

Each movement is logged just like in the main part but compiled into a single bonus output file bonus.txt.

Conclusion

The **Gold Trail** project demonstrates a fusion of **algorithmic problem-solving**, **software engineering design**, and **visualization techniques**. Through modular class design and the use of advanced data structures, it effectively tackles both a classic shortest path problem and a complex route optimization task.

The sequential pathfinding leverages Dijkstra's algorithm to guarantee optimal subpaths, while the bonus part applies intelligent heuristics to find a near-optimal full tour across all objectives. The clear separation of responsibilities between Tile, MapHandler, PathFinder, and Bonus classes enhances code clarity and maintainability.

In short, the knight doesn't just wander — he plans, optimizes, and triumphs. The application showcases not only technical proficiency but also a strong grasp of real-world algorithm application and time-constrained optimization.