Mehmet Arda Kutlu

Assignment 3 Report

Introduction

About the Project: Unlike the previous assignments, Gold Trail: The Knight's Path focuses on algorithmic problem solving rather than interactive gameplay. In this project, a virtual map is constructed using three different types of tiles, with a knight positioned on one of them and several coins scattered across the grid. Both the knight and the coins have predefined positions that are obtained through the input files. The primary objective is to calculate the shortest path one by one from the knight to each accessible coin individually. Additionally, the program also animates the movement of the knight on the map using StdDraw if the -draw flag is used. In the bonus part, instead of finding the paths individually, the algorithm computes the full path that the knight can follow to collect all the accessible coins and then return to its initial position in the lowest possible total cost.

Components of the Code

Tiles: Tiles are the components that make up the map. They have an important role in the algorithm since the knight moves across them. Each movement of the knight has an associated cost depending on both the tile he moved from and the tile he moved to. There are three types of tiles: grass tiles that have relatively low movements costs, sand tiles that have higher costs compared to grass tiles, and impassable tiles that have infinite costs, which means that the knight cannot cross them.

Coins: Coins are the stationary objectives that the knight should collect by reaching their positions. The important point is that not all the coins are reachable, since some of them may be completely surrounded by impassable tiles. The algorithm is designed to identify such coins and skip them during pathfinding.

Knight: The knight represents the main character that tries to collect all reachable coins. In the standard part, he collects coins one by one, starting from the initial position to each target coin using the shortest path. In bonus part, it follows the optimal round-trip path that has the lowest possible movement cost.

Class Diagrams

The code has eleven classes, two of which are main classes that runs the code: Main for the standard part and Bonus for the bonus part. Other classes undertake three primary tasks: storing the game components, calculating the paths and interacting with input and output files.

Knight Class: Represents the knight character that is tasked with collecting all reachable coins. It stores the knight's properties which are his row and column indices on the map.

Table 1: UML Diagram of the Knight Class

Knight	
- col: int - row: int	x coordinate of the knight on the map. y coordinate of the knight on the map.
+ Knight (col: int, row: int) + getColNum(): int + setColNum(colNum: int): void + getRowNum(): int + setRowNum(rowNum: int): void	Constructs a knight object with given row and column indices. Returns the knight's current x position. Sets the knight's x position. Returns the knight's current y position. Sets the knight's y position.

Coin Class: Represents the objectives that the knight has to reach. It stores the knight's properties such as its x and y position and collection status.

Table 2: UML Diagram of the Coin Class

Coin	
- col: int - row: int - isCollected: boolean	x coordinate of the coin on the map. y coordinate of the coin on the map. Whether the knight has collected the coin. (Whether the coin will be displayed)
+ Coin (col: int, row: int) + getColNum(): int + getRowNum(): int + isCollected(): boolean + setCollected(isCollected: boolean): void	Constructs a coin object with given row and column indices. Returns the coin's x position. Returns the coin's y position. Returns the current collection status of the coin (whether it is collected) Sets the collection status of the coin.

Tile Class: Represents the tiles in the map (grid). Stores the tile properties, which are the tile position (row and column indices), type of the tile (sand, grass, or impassable), neighboring tiles of each tile as an ArrayList, whether the knight passed the tile, and for the bonus part, color of the dot that marks the tile as passed.

Table 3: UML Diagram of Tile Class

Tile	
- column: int - row: int - type: int - adjacentTiles: ArrayList<Tile> - isMarked: boolean - markColor: Color	Column index of the tile. Row index of the tile. Type of the tile (0 is grass, 1 is sand, 2 is impassable). Neighboring tiles of the tile. Whether the knight passed the tile. Color of the dot that marks the tile as passed. Only needed in the bonus part.

+ Tile (column: int, row: int, type: int)	Constructs a tile object at specified positions with specified type.
+ getColumn(): int	Returns the x position of the tile.
+ getRow(): int	Returns the y position of the tile.
+ getType(): int	Returns the type of the tile.
	Sets the collection status of the coin.
+ isEqual(tile: Tile): boolean	Checks if the given tile is the same with the tile by looking both the coordinates and the types.
+ addAdjacentTile(tile: Tile): void	Adds a neighboring tile to **adjacentTiles** ArrayList.
+ getAdjacentTiles(): ArrayList<Tile>	Returns the neighboring tiles of the tile as an ArrayList.
+ isMarked(): boolean	Returns the marking status of the tile.
+ setMarked(marked: Boolean): void	Sets the marking status of the tile.
+ getMarkColor(): Color	Returns the color of the dot. Only used in the bonus part.
+ setMarkColor(markColor: Color): void	Sets the dot color of the tile. Only used in the bonus part.

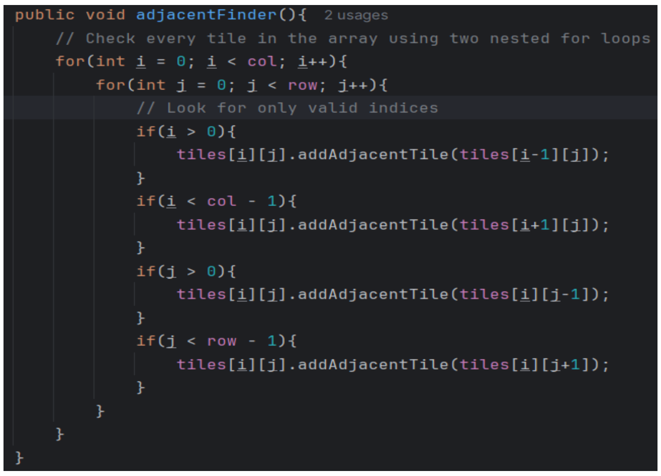
Map Class: Stores the primary components (tiles, coins and the knight). Provides methods that enable other classes to interact with map objects, and visualize the program.

Table 4: UML Diagram of the Map Class

Map	
- col: int	Number of tile columns in the map. Column and row number are determined by the size of the tile grid that is given in the first line at *mapData.txt*.
- row: int	Number of tile rows in the map. Determined by the row number of the grid.
- tiles: Tile[][]	2D array that aims to store the tiles similar to their positions in the map and make finding adjacent tiles easier.
- coins: ArrayList<Coin>	Stores the coins.
- knight: Knight	The knight in the map.
- canvasHeight: int - canvasWidth: int	Width and height of the canvas is adjusted according to the row and column number.
+ Map (col: int, row: int)	Constructs a map object with given row and column number of tiles. Defines the size of the tiles array and sets the canvas size.
+ getCanvasWidth(): int	Returns the canvas width.
+ getCanvasHeight: int	Returns the canvas height.
+ fillTiles(colNum: int, rowNum: int, tile: Tile): void	Puts a tile object to the tiles array which was initially empty. Tiles are added one by one using the information in *mapData.txt*.

+ getTiles(): Tile[][]	Returns the tiles array.
+ getKnight(): Knight	Returns the knight object.
+ addCoin(coin: Coin): void	Adds a coin object to the coins array list which was initially empty. Coins are added one by one using the information in *objectives.txt*.
+ getCoins(): ArrayList<Coin>	Returns the coins ArrayList.
+ xCenterFinder(colNum: int): double	Takes the column number as input and calculates the corresponding x coordinate in the canvas. Facilitates drawing objects.
+ yCenterFinder(rowNum: int): double	Takes the row number as input and calculates the corresponding y coordinate in the canvas. Facilitates drawing objects.
+ adjacentFinder(): void	Checks all the tiles in the tiles array and finds the neighboring tiles for each of them. Harnesses 2D array mechanism.
+ draw(): void	Draws the components to the canvas for the standard part.
+ isKnightOnCoin(objectiveNum: int): Boolean	Checks if the knight is on the same tile with the coin at specified index of an objective list Only used in the bonus part.
+ knightIsOn(knight: Knight): Tile	Returns the tile that the knight is currently standing.
+ coinIsOn(coin: Coin): Tile	Returns the tile that the given coin is standing on.
+ drawBonus(): void	Draws the components to the canvas for the bonus part. Handles path visualizing differently from the standard draw method.

Additional Notes for the Map Class: The positions of the tiles in the **tiles** array are similar with their positions in the grid. **adjacentFinder** method finds the adjacent tiles by checking the valid consecutive indices in the array and adds the tiles at these indices to **adjacentTiles** ArrayList for each of the tiles.



```
public void adjacentFinder(){  2 usages
    // Check every tile in the array using two nested for loops
    for(int i = 0; i < col; i++){
        for(int j = 0; j < row; j++){
            // Look for only valid indices
            if(i > 0){
                tiles[i][j].addAdjacentTile(tiles[i-1][j]);
            }
            if(i < col - 1){
                tiles[i][j].addAdjacentTile(tiles[i+1][j]);
            }
            if(j > 0){
                tiles[i][j].addAdjacentTile(tiles[i][j-1]);
            }
            if(j < row - 1){
                tiles[i][j].addAdjacentTile(tiles[i][j+1]);
            }
        }
    }
}
```

Figure 1: adjacentFinder method

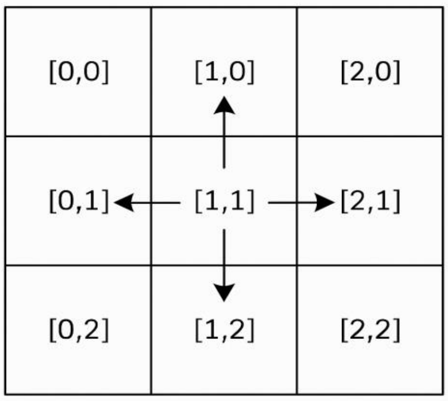


Figure 2: Searching for neighboring tiles

Additionally, there is a distinction in the visualization of the path between standard and the bonus part. In the standard part, the path goes from the initial position of the knight to the target objective, and the passed tiles are marked with a red dot at their center. When the knight reaches the coin, the coin and all of the marks should be removed from the map. However, in the bonus part, the marks should change color instead of disappearing when the knight collects a coin. Because of this distinction, visualization of the program is handled by two different draw methods.

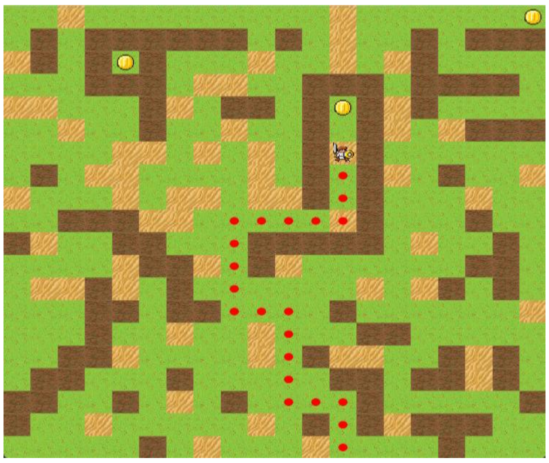


Figure 3: Visualization in standard part

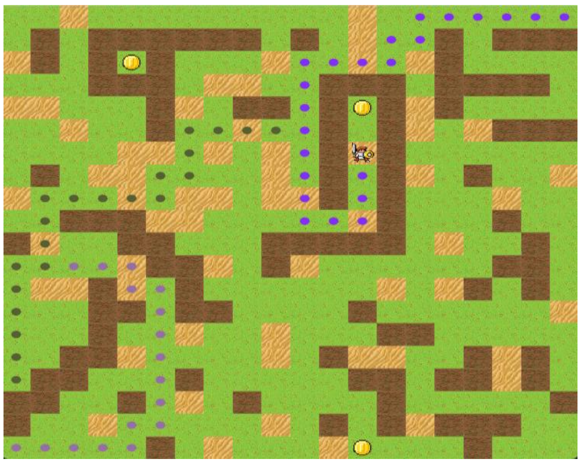


Figure 4: Visualization in bonus part

Reader Class: Reader class handles reading map and coin data from *mapData.txt* and *objectives.txt*, respectively. It provides methods to read integers from the files. *travelCosts.txt* file is not included because PathFinder class reads this file in its constructor.

Table 5: UML Diagram of the Reader Class

Reader	
- mapData: File - objectives: File mapReader: Scanner - objectiveReader: Scanner	File object for *mapData.txt*. File object for *objectives.txt*. Scanner object to read data from *mapData.txt*. Scanner object to read data from *objectives.txt*.
+ Reader (mapDataName: String, objectivesName: String) + readMap(): int + isMapInfoFinished(): boolean + readObjectives(): int + isObjectiveInfoFinished(): boolean + closeScanners(): void	Constructor of the class. File names are provided in the main class from the console. Reads the next integer from *mapData.txt*. Checks if there is more map data available to read. Reads the next integer from *objectives.txt*. Checks if there is more objective data available to read. Closes both scanners to avoid unexpected behavior.

Writer Class: Responsible for writing the string output to the output file.

Table 6: UML Diagram of the Writer Class

Writer	
- fileWriter: FileWriter - outputWriter: BufferedWriter	FileWriter object to use in BufferedWriter. BufferedWriter to write content.
+ Writer (filename: String) + closeBuffer(): void + write(toWrite: String): void	Creates a writer object that writes output to given output file. Closes BufferedWriter to avoid unexpected behavior. Writes the given string to the output file.

PathFinder Class: Responsible for determining the shortest path between two tiles on the map using Dijkstra's algorithm. It stores predefined movement costs between tile pairs in a hash map for efficient lookup. When a path is needed, the class calculates the least costly route from a starting tile to a target tile, considering only valid moves and tile types. It also provides methods to compute the total cost of a given path by summing the movement costs between each consecutive pair of tiles.

Table 7: UML Diagram of PathFinder Class

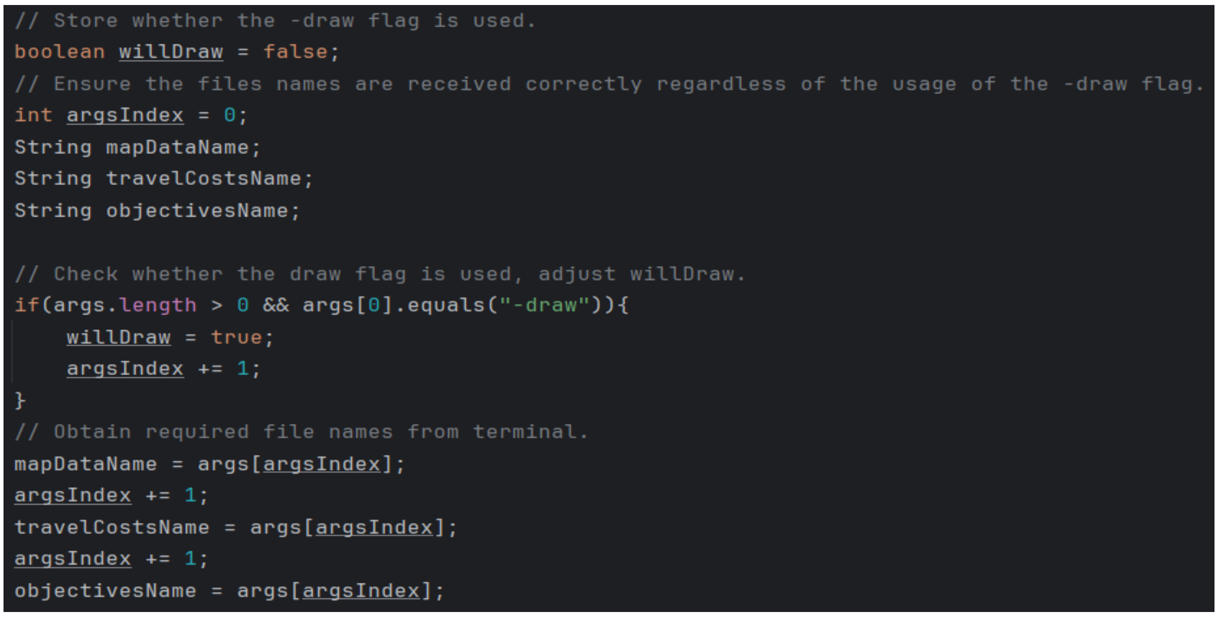
PathFinder	
- TravelCosts: HashMap<String, Double>	Stores the travel cost between two adjacent tiles using a string key in both ways (both tile1 -> tile2 & tile2 -> tile1).
+ PathFinder (travelCostName: String) + costCalculator(tile1: Tile, tile2: tile): double + algorithm(start: Tile, objective: Tile): ArrayList<Tile> + findTotalCost(path: ArrayList<Tile>): double	Constructs a PathFinder object. Fills **TravelCosts** with the data obtained from *travelCosts.txt*. Returns the movement cost between two tiles, or -1 if the movement is impossible. Uses Dijkstra's algorithm to compute the shortest path between two tiles. Calculates the total cost of moving along a path.

ShortestRoute Class: The ShortestRoute class implements the algorithm to solve the Traveling Salesman Problem (TSP) in the form of the knight collecting all reachable coins and returning to the starting position. It precomputes the shortest paths and associated costs between every pair of relevant tiles (the starting point and each coin) using the PathFinder class. Then, it applies a dynamic programming approach (Held-Karp algorithm) to determine the optimal visiting order that minimizes the total travel cost. The class returns the complete tour path as an ordered list of tiles representing the knight's movements.

Table 8: UML Diagram of the ShortestRoute Class

ShortestRoute	
+ ShortestRoute ()	Default constructor of the class.
+ findShortestTour(start: Tile, objectives: List<Tile>, pathFinder: PathFinder)	Calculates the path with lowest total cost that starts with the knight's initial position, visits all the reachable coins and returns to the starting position.

Main Class: The Main class is responsible for handling the standard part of the project. It manages the setup and execution of the knight's coin collection process. It takes the draw flag and the input file names as command-line arguments, reads input data files using the Reader class, and constructs the map, knight, and coin objects. It then initializes the PathFinder and AlgorithmRunner classes to perform pathfinding and execute the algorithm that guides the knight to each reachable coin one by one. If the draw flag is used and there is at least one reachable coin, the class also enables animation of the knight's movement across the map. Finally, it writes the step-by-step output to *output.txt* file.



```java
// Store whether the -draw flag is used.
boolean willDraw = false;
// Ensure the files names are received correctly regardless of the usage of the -draw flag.
int argsIndex = 0;
String mapDataName;
String travelCostsName;
String objectivesName;

// Check whether the draw flag is used, adjust willDraw.
if(args.length > 0 && args[0].equals("-draw")){
    willDraw = true;
    argsIndex += 1;
}
// Obtain required file names from terminal.
mapDataName = args[argsIndex];
argsIndex += 1;
travelCostsName = args[argsIndex];
argsIndex += 1;
objectivesName = args[argsIndex];
```

Figure 5: Receiving the command-line arguments

Bonus Class: The Bonus class is responsible for executing the bonus part of the project. It functions similarly to the Main class in terms of setup and initialization, but there are several key differences in how the algorithm is executed. Instead of finding and following individual shortest paths to each coin, the Bonus class uses the more advanced ShortestRoute class to compute a complete round-trip path that allows the knight to collect all reachable coins and return to the starting position with the minimum total movement cost. It also replaces the standard **run** and **draw** methods with **runBonus** and **drawBonus** methods from the AlgorithmRunner and Map classes, respectively, in order to support continuous path visualization with different colors specific to the bonus mode. Output of the bonus code is written to *bonus.txt* instead of *output.txt*.

Algorithm

To achieve computing the shortest path between two tiles on the map, the ideal algorithm is the Dijkstra’s algorithm since the movement costs have positive values. It simply finds the shortest path from a starting node to the target node in the graph where all the edges have non-negative weights (costs). In the assignment, the tiles are the nodes in the graph, and movements between the tiles are the weighted edges. The weights (movement costs) between tiles are obtained from *travelCosts.txt* file, and are stored in a hash map for fast access. Dijkstra’s algorithm calculates the path in 5 steps:

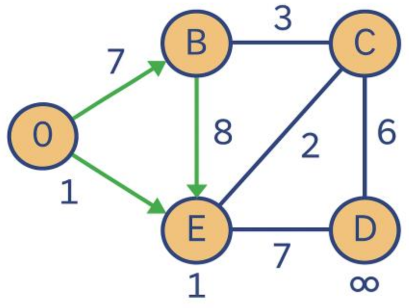


Figure 6: Dijkstra’s algorithm on a simple graph

Initialization: A priority queue (min-heap) is initialized with the starting tile, and a hash map is used to keep track of the current minimum cost to reach each tile. The starting tile’s cost is set to 0, and all others are initialized with positive infinity.

Exploration: The algorithm repeatedly extracts the tile with lowest known cost from the priority queue and evaluates all of its adjacent, passable tiles. For each neighbor, it calculates the cost of reaching that neighbor through the current tile.

Cost Update: If a shorter path to a neighbor is found, the neighbor's cost is updated and it is added to the priority queue for further exploration. A separate map is used to keep track of the previous tile for each visited tile, allowing the final path to be reconstructed once the destination is reached.

Termination: The algorithm stops when the destination tile is reached or when all reachable tiles have been explored and the destination is found to be unreachable.

Path Reconstruction: Once the shortest path is found, it is reconstructed by backtracking from the destination tile to the starting tile using the **previousTile** map, and then reversing the result to get the forward path.

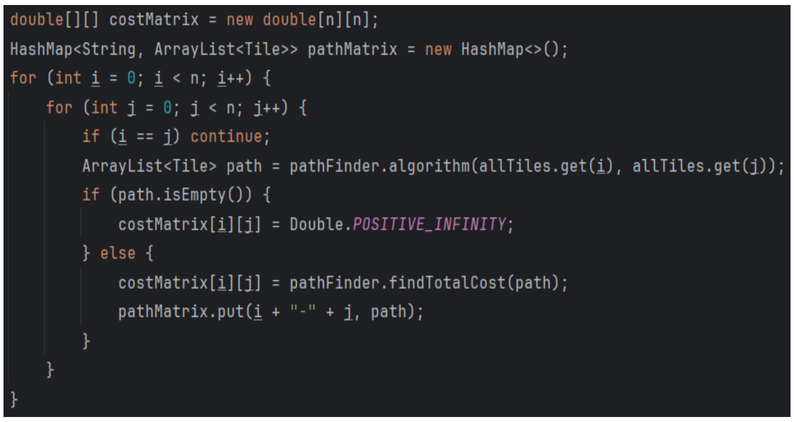
This approach ensures that the knight always follows the least costly valid path to reach a coin. The use of Dijkstra’s algorithm is especially appropriate here because it handles varying movement costs effectively and guarantees optimal paths in graphs with non-negative edge weights (movement costs), which matches the requirements of the tile-based map structure used in this project.

Bonus

In the bonus part, we were asked to find the most efficient path that the knight can follow to collect all the coins without any order and then return to his initial position with lowest possible total cost, which is known as **Traveling Salesman Problem (TSP).** To efficiently solve the problem within three seconds, I implemented the Held-Karp dynamic programming algorithm, which is a well-known approach for solving the TSP.

Algorithm Overview

Path Cost Matrix: The algorithm first computes the shortest path and movement cost between every pair of nodes (the knight and each coin) using Dijkstra's algorithm from the PathFinder class. These paths and their costs are stored in a 2D array **costMatrix[i][j]**, which stores the travel cost from node i to node j. A **pathMatrix** hash map that stores the actual sequence of tiles used to get from one node to another.

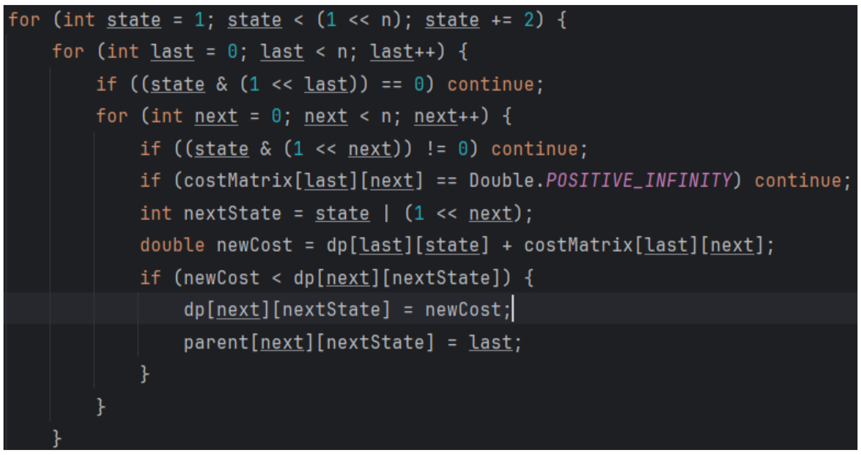


```java
double[][] costMatrix = new double[n][n];
HashMap<String, ArrayList<Tile>> pathMatrix = new HashMap<>();
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == j) continue;
        ArrayList<Tile> path = pathFinder.algorithm(allTiles.get(i), allTiles.get(j));
        if (path.isEmpty()) {
            costMatrix[i][j] = Double.POSITIVE_INFINITY;
        } else {
            costMatrix[i][j] = pathFinder.findTotalCost(path);
            pathMatrix.put(i + "-" + j, path);
        }
    }
}
```

Figure 7: Path cost matrix

Dynamic Programming Setup: To solve the TSP, I used a dynamic programming table dp[i][S], where i is the index of the last visited node and S is a bitmask representing the set of visited nodes. Each entry stores the minimum cost to reach node i after visiting all nodes in subset S.

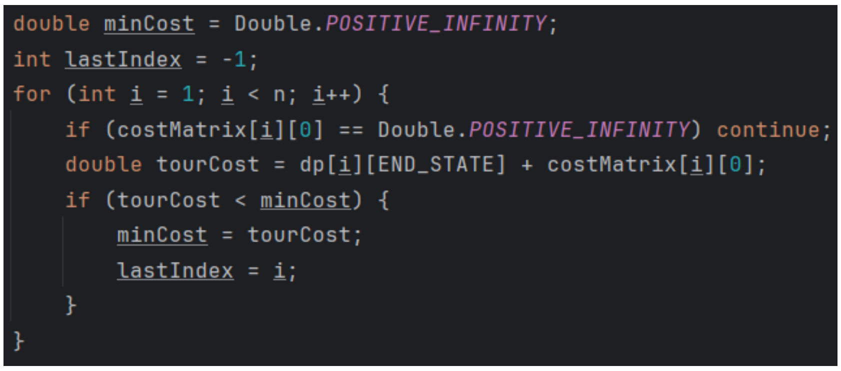
Recurrence Relation: For each state S and node i in that state, the algorithm considers all possible next nodes j not yet visited and updates the DP table. It also keeps track of the parent node for each state in a separate parent table to allow path reconstruction later.



```java
for (int state = 1; state < (1 << n); state += 2) {
    for (int last = 0; last < n; last++) {
        if ((state & (1 << last)) == 0) continue;
        for (int next = 0; next < n; next++) {
            if ((state & (1 << next)) != 0) continue;
            if (costMatrix[last][next] == Double.POSITIVE_INFINITY) continue;
            int nextState = state | (1 << next);
            double newCost = dp[last][state] + costMatrix[last][next];
            if (newCost < dp[next][nextState]) {
                dp[next][nextState] = newCost;
                parent[next][nextState] = last;
            }
        }
    }
```

Figure 8: Considering next nodes

Tour Completion: After the DP table is fully computed, the algorithm finds the minimum-cost path that returns to the starting point (node 0). It selects the best path by choosing the last node that gives the lowest total cost when returning to the start.



```
double minCost = Double.POSITIVE_INFINITY;
int lastIndex = -1;
for (int i = 1; i < n; i++) {
    if (costMatrix[i][0] == Double.POSITIVE_INFINITY) continue;
    double tourCost = dp[i][END_STATE] + costMatrix[i][0];
    if (tourCost < minCost) {
        minCost = tourCost;
        lastIndex = i;
    }
}
```

Figure 9: Finding the path with minimum cost

Path Reconstruction: The algorithm then reconstructs the full tour by backtracking through the parent table and concatenating the corresponding tile sequences from the **pathMatrix**.

Since the number of coins in the project is limited (max 20), this algorithm performs efficiently in practice and guarantees finding the shortest possible route instead of approximating. This approach successfully reduces the total travel cost compared to visiting coins one by one.