Bogazici University

Introduction to Object Oriented Programming

CMPE160

Assignment 2

Author:
Ibrahim Berkay Ceylan
Student ID: 2023400327

April 4, 2024

Contents

1 Introduction

This report presents the development of a Turkey city navigation application, a project assigned in the CMPE160 course in Bogazici University. The primary objective of the application is to efficiently calculate and display the shortest path between two cities within Turkey. This project makes use of the Java programming language with the StdDraw library to model cities, their connections, etc.

The main challenge of this project lies in the application of an efficient path-finding algorithm, which in this project is Dijkstra's Algorithm for real world navigation. The implementation of this algorithm not only serves a practical purpose but demonstrates the use of algorithmic thinking and object-oriented programming principles. The project adheres to given specifications and handles various scenarios including invalid city names, unreachable city pairs and navigation to the same city.

The following sections will detail the implementation along with outputs generated in each scenario. This report aims to showcase not just the functionality of the developed application, but also the best practices in Java programming and software development.

2 Implementation Details

In the application, several classes are utilized to manage different aspects of the functionality. The classes include Main, City, PathFinder, MapManager, MapDrawer. Each class has a specific role, the following sections describes each class with their corresponding UML Class diagrams.

- **Main:** Serves as the entry point of the application. Responsible for handling user input, handling file reading, operations of other classes, etc.

- **City:** Represents a city with attributes like name, coordinates, and connections to other cities.

City
- cityName: String - x: int - y: int - index: int - connections: ArrayList¡City¿
+ getCityName() : String + getX(): int + getY(): int + getIndex(): int + getConnections(): ArrayList<City> + connect(city: City): void

- **PathFinder:** Implements Dijkstra's Algorithm to find the shortest path between two cities.

PathFinder
- mapManager: MapManager - mapDrawer: MapDrawer
+ findOptimalPath(startCity: City, targetCity: city) - initializeDistanceArray(distance: double[], startCity: city) - findPathLogic(targetCity: City, distance: double[], previous: City[], visited: boolean[]): void - findNearestCity(distance: double[], visited: boolean[]): City - updateDistances(city: City, distance: double[], previous: City[]) : void - processPathResults(distance: double[], previous City[], targetCity: City): void - calculateDistance(city1: City, city2: City): double - buildPath(previous: City[], targetCity: City): ArrayList<City> - reversePath(path: ArrayList<City>): void - printDistanceAndPath(pathToTarget: ArrayList<City >): void - calculateTotalDistance(pathToTarget: ArrayList<City>): double

- **MapManager:** Manages city data, reading city information, connection information from files.

MapManager
- cities: ArrayList<City>
+ readCitiesFromFile(filename: String): void
+ readConnectionsFromFile(filename: String): void
+ findCityByName(cityName: String): City
+ getCities(): ArrayList<City>

- **MapDrawer:** Responsible for drawing the map, cities, roads and optimal path on the canvas.

MapDrawer
- WIDTH: int
- HEIGHT: int
- SCALE: int
- font: Font
+ initializeCanvas(): void
+ drawMap(): void
+ drawCities(cities: ArrayList<City>): void
+ drawCity(city: City): void
+ drawRoads(cities: ArrayList<City>): void
- drawPath(city1: City, city2: City): void
+ drawOptimalPath(path: ArrayList<City>): void

The core pathfinding functionality is encapsulated in the PathFinder class, which implements Dijkstra's algorithm.

2.1 Explanation of the Path-finding Algorithm

Dijkstra's Algorithm finds the shortest path between nodes in a graph. The core idea of the algorithm is to keep track of the shortest known distance to each node and to update these distances based on new information acquired while traversing the graph.

Here is a detailed explanation of how the algorithm works:

1. Initialization
   - (a) Set the distance to the initial node to 0 and to all other nodes to infinity.
   - (b) Keep a structure to select the node with the smallest known distance
   - (c) Mark all nodes as unvisited
2. Visiting Nodes
   - (a) Visit the unvisited node with the smallest distance.
   - (b) Mark the current node as visited.
   - (c) For each unvisited neighbour of the current node:
     - i. Calculate the distance from the current node to the neighbor
     - ii. If this distance is less than the previously known distance to the neighbor, update the distance and record the current node as 'previous' on the path to the neighbor.
3. Repetition
   - Repeat the process until all possible nodes are visited or destination path is found.
4. Reconstructing the Path
   - Once the destination node's shortest path is known (or all nodes are visited), reconstruct the shortest path from the destination to the initial node using the recorded "previous nodes".

Now we present the pseudocode for the pathfinding algorithm implemented in the 'PathFinder' class. The algorithm is utilized to determine the shortest path between two cities in a network of roads.

Algorithm 1 Optimal Path Finding using Dijkstra's Algorithm

```
Data: Cities, Connections, startCity, targetCity
Result: Optimal path from startCity to targetCity
Function findOptimalPath(startCity, targetCity):
    Initialize distance array with Infinite for all cities, 0 for startCity
    Initialize previous array with null for all cities
    Initialize visited array with False for all cities

    while there are unvisited cities do
        nearestCity = find nearest unvisited city
        if nearestCity is null or nearestCity is targetCity then
            break
        end
        Mark nearestCity as visited
        for each neighbor of nearestCity do
            alt = distance[nearestCity] + distance between nearestCity and neighbor
            if alt < distance[neighbor] then
                Update distance[neighbor] and previous[neighbor]
            end
        end
    end
    pathToTarget = build path from previous array
    if pathToTarget is not empty then
        Draw path and print total distance
    else
        Print no path found message
    end
```

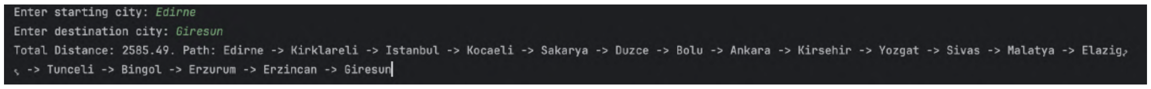
3 Test Cases and Outputs

In this section, we evaluate the functionality and robustness of the Turkey City Navigation application through the means of test cases. These tests are designed to cover various scenarios, including valid routes between different cities, handling of invalid city inputs, routes to the same city, and cases where no path exists between two cities.

3.1 Valid Route Between Two Different Cities

This test case demonstrates the application's intended non-trivial function: Identifying the shortest path between two distinct cities. It validates the implementation of Dijkstra's algorithm in finding the most efficient route.

For instance, when a user inputs the route from "Edirne to Giresun," the application processes this input to calculate and display the shortest path.



```
Enter starting city: Edirne
Enter destination city: Giresun
Total Distance: 2585.49. Path: Edirne -> Kirklareli -> Istanbul -> Kocaeli -> Sakarya -> Duzce -> Bolu -> Ankara -> Kirsehir -> Yozgat -> Sivas -> Malatya -> Elazig↵
↳ -> Tunceli -> Bingol -> Erzurum -> Erzincan -> Giresun
```

Figure 1: Console output for valid route.

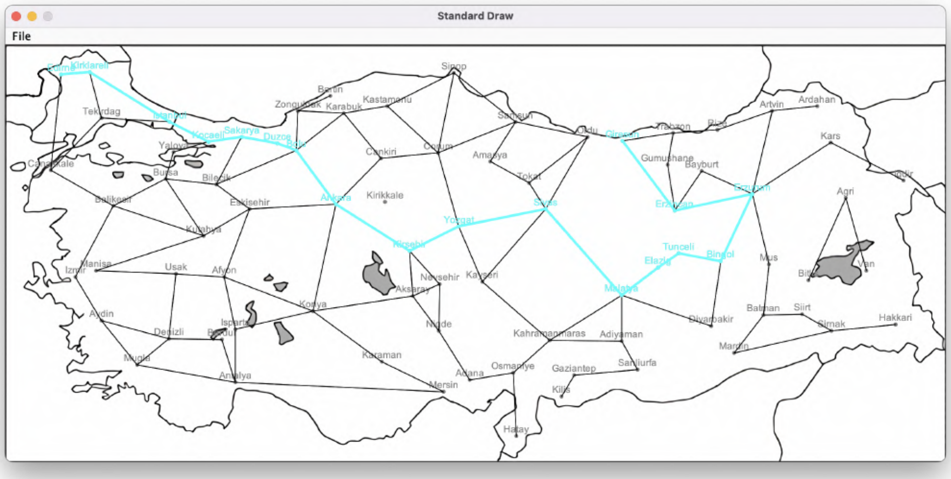
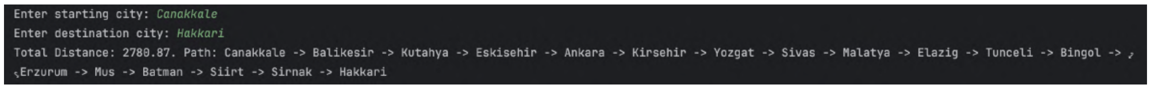


Figure 2: Path drawing for valid route.



```
Enter starting city: Canakkale
Enter destination city: Hakkari
Total Distance: 2780.87. Path: Canakkale -> Balikesir -> Kutahya -> Eskisehir -> Ankara -> Kirsehir -> Yozgat -> Sivas -> Malatya -> Elazig -> Tunceli -> Bingol -> ↲
↳Erzurum -> Mus -> Batman -> Siirt -> Sirnak -> Hakkari
```

Figure 3: Console output for another valid route.

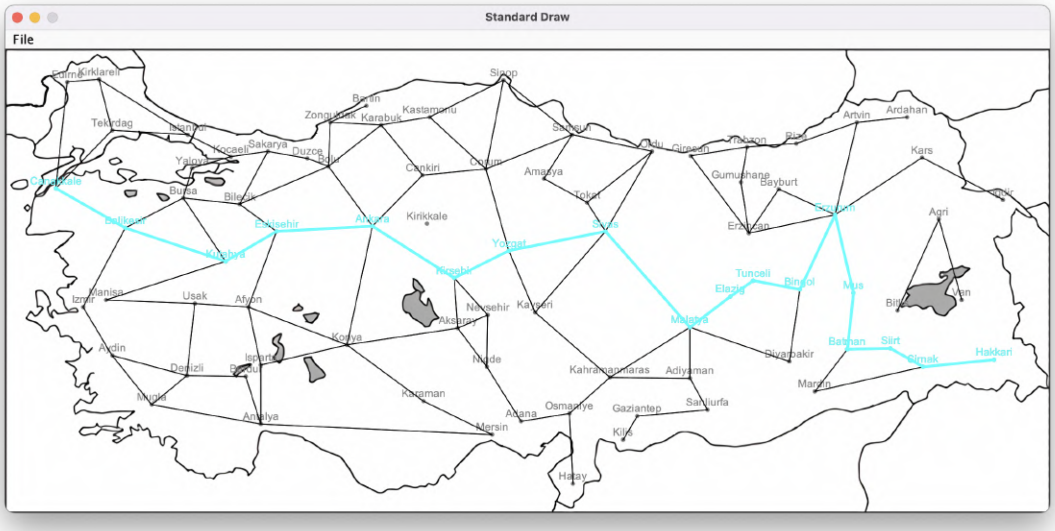
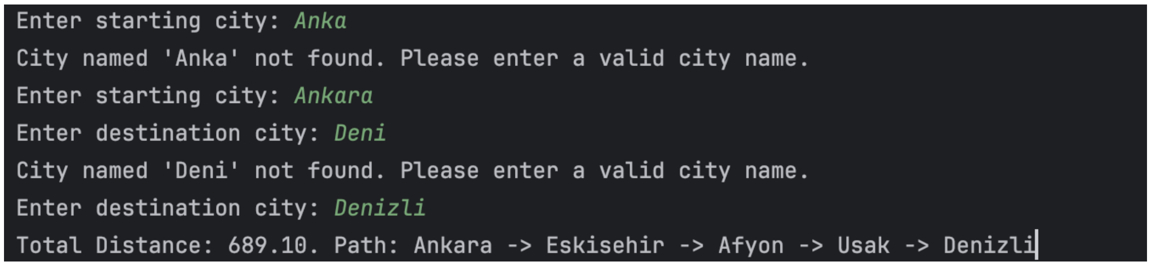


Figure 4: Path drawing for another valid route.

3.2 Invalid City Names

This test assesses the program's ability to handle invalid city name inputs gracefully.



```
Enter starting city: Anka
City named 'Anka' not found. Please enter a valid city name.
Enter starting city: Ankara
Enter destination city: Deni
City named 'Deni' not found. Please enter a valid city name.
Enter destination city: Denizli
Total Distance: 689.10. Path: Ankara -> Eskisehir -> Afyon -> Usak -> Denizli
```

Figure 5: Case when invalid city names are inputted.

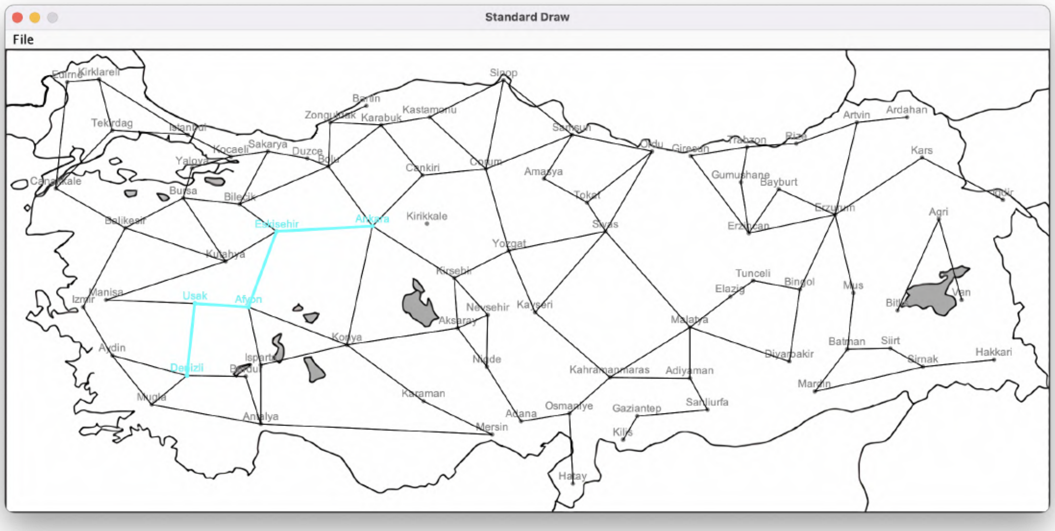
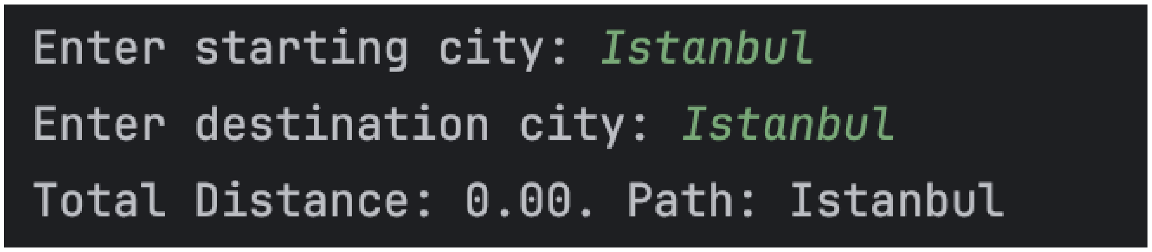


Figure 6: Path drawing of a valid route.

3.3 Route to the Same City

This scenario tests the application's response when the starting and destination cities are the same.



```
Enter starting city: Istanbul
Enter destination city: Istanbul
Total Distance: 0.00. Path: Istanbul
```

Figure 7: Output when starting city and destination are the same.

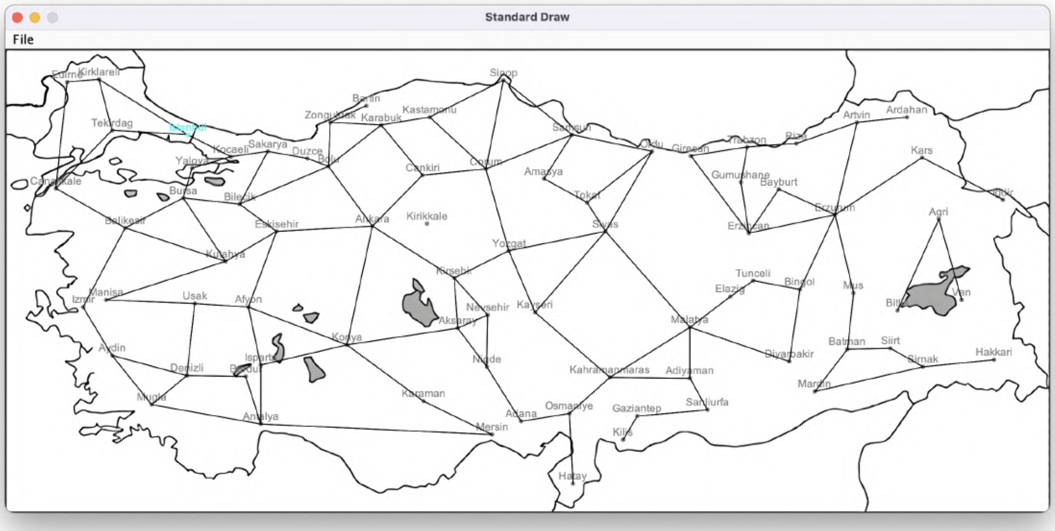
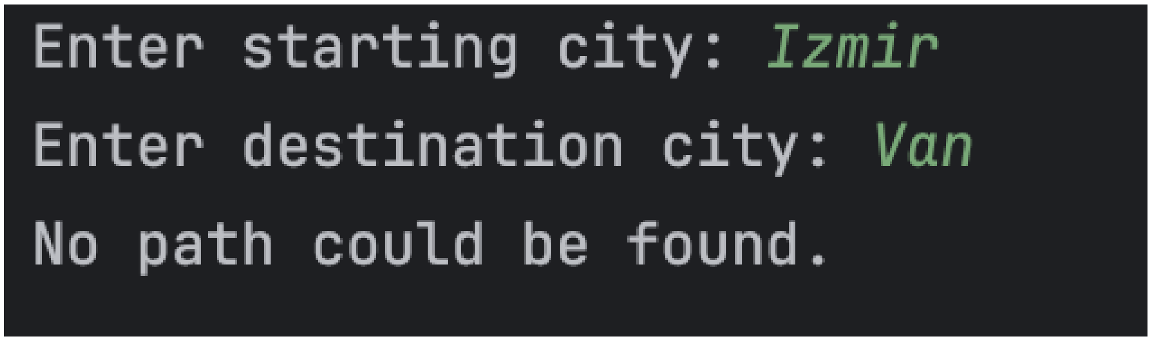


Figure 8: Canvas when starting and destination cities are the same.

3.4 Unreachable City Pairs

This test case evaluates how the application behaves when there is no path available between the chosen cities, no graphical output is produced in this test case.



```
Enter starting city: Izmir
Enter destination city: Van
No path could be found.
```

Figure 9: Output when there is no path available between the chosen cities.