Code Overview:

The provided Java code is for a program called "Turkey Navigation" which reads city coordinates and connections from files, prompts the user for input, calculates the shortest path between two cities, and visualizes it on a map.

Key Components:

Main Method:

Reads city coordinates and connections from files.
Prompts the user for input (starting and destination cities).
Calculates the shortest path between the provided cities.
Draws the map with cities, connections, and the shortest path.
City Class:

Represents a city with name, x and y coordinates, and connections to other cities.
Provides methods to calculate distance between cities and to draw the city and roads.
Shortest Path Finder:

Uses Dijkstra's algorithm to find the shortest path between two cities.
Constructs a map with distances between cities.
Implements Dijkstra's algorithm to find the shortest path.
Reconstructs the shortest path and prints the total distance and path.
Utility Methods:

isCityExisting: Checks if a city exists in the list of cities.
indexFinder: Finds the index of a city in the list of cities.
drawShortestPath: Draws the shortest path between cities on the map.
How It Works:

The program starts by reading city coordinates and connections from files.
It prompts the user to input the starting and destination cities.
Using Dijkstra's algorithm, it finds the shortest path between the provided cities.
It then visualizes the map with cities, connections, and the shortest path.
Finally, it prints the total distance of the shortest path and the path itself.
Purpose:

The purpose of this program is to assist users in navigating between cities in Turkey by finding the shortest path between two cities and visualizing it on a map.
It can be helpful for travelers or logistics planners to determine the most efficient route between cities.

Dijkstra's Algorithm:

Dijkstra's algorithm is a widely used algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. Here's how it works:

Initialization:

Initialize an array shortestDistances[] where each element represents the shortest distance from the starting node to that node.
Initialize all distances to a maximum value (except the starting node's distance, which is set to 0).
Initialize a boolean array visited[] to keep track of which nodes have been visited.
Initialize a priority queue or use another data structure to keep track of the next closest node to explore.
Iteration:

Repeat the following steps until all nodes have been visited:
Find the node with the shortest distance among the unvisited nodes.
Mark this node as visited.
Update the distances of its neighboring nodes if a shorter path is found through the current node.
Update the priority queue or data structure accordingly.
Termination:

Once all nodes have been visited, the shortest path to each node from the starting node is determined.
Path Reconstruction:

After the algorithm finishes, the shortest path to any node can be reconstructed by backtracking through the shortestDistances[] array.
Explanation in the Code:

Map Construction:

In the provided code, the map between cities is represented by a 2D array map[][], where map[i][j] represents the distance between city i and city j. This distance is calculated using the distanceCalculator method.
The map[][] array is filled based on the connections between cities read from the input files.
Initialization:

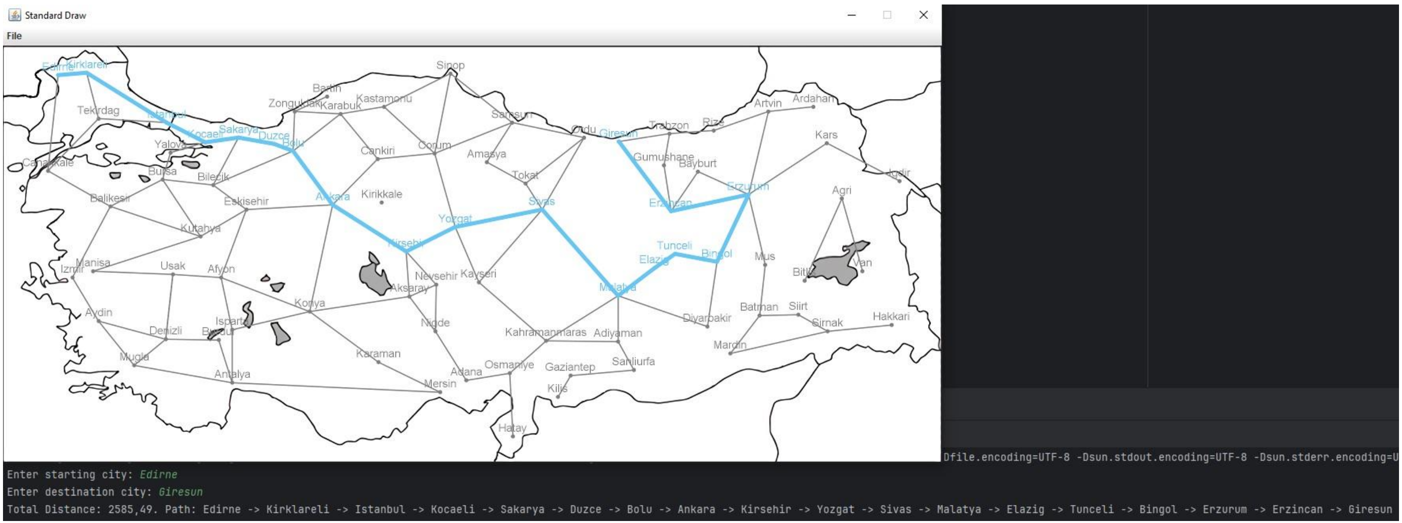
Arrays shortestDistances[] and visited[] are initialized to keep track of shortest distances and visited cities, respectively.
Initially, all distances except the starting city's distance are set to a maximum value. The starting city's distance is set to 0.
Iteration:

The algorithm iterates over all unvisited cities. In each iteration, it finds the closest unvisited city using the shortestDistances[] array.
For each unvisited neighbor of the current city, if a shorter path is found through the current city, its distance is updated.
The algorithm continues until all cities are visited or until the destination city is reached.
Path Reconstruction:

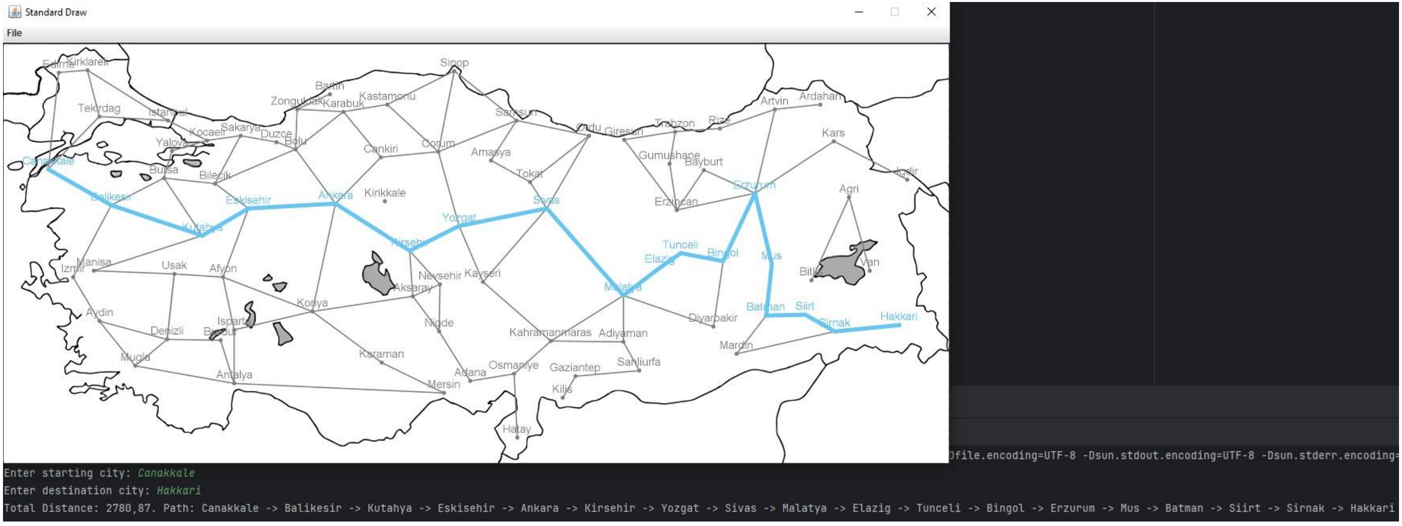
Once the algorithm finishes, the shortest path is reconstructed by backtracking through the shortestPathCities[] array, which keeps track of the previous city on the shortest path.

Pseudocode:

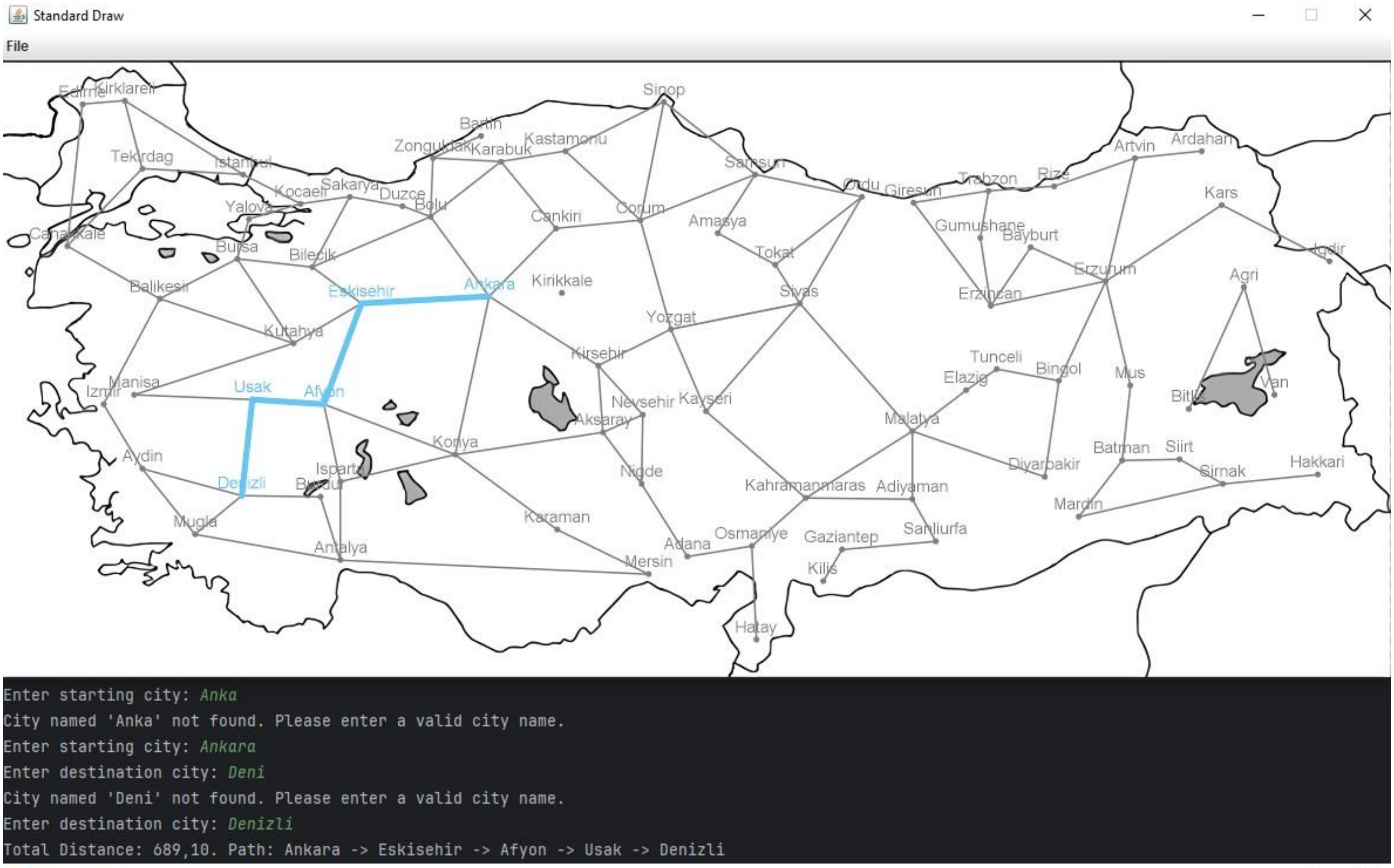
```
Method: shortestPathFinder
  Input: Graph G with cities and connections,
      startingCity s,
      destinationCity t
  Output: Shortest path from s to t
  // Initialize variables
  numberOfLocations = number of cities in G
  map = 2D array of doubles with dimensions [numberOfLocations][numberOfLocations]
  maxVal = maximum value for distance

  // Initialize arrays for shortest distances, visited cities, and shortest path cities
  shortestDistances = array of doubles with length numberOfLocations
  fill shortestDistances with maxVal
  shortestDistances[indexFinder(cities, startingCity)] = 0
  visited = array of booleans with length numberOfLocations
  fill visited with false
  shortestPathCities = array of integers with length numberOfLocations
  fill shortestPathCities with -1

  // Construct the map with distances between cities
  For each city in G:
    cityIndex = indexFinder(cities, city)
    For each neighbor in city.connections:
      neighborCity = null
      For j from 0 to size of cities - 1:
        If cities[j] equals neighbor:
          neighborCity = cities[j]
          neighborCityIndex = j
          break
      If neighborCity is not null:
        distance = distance between city and neighborCity
        map[cityIndex][neighborCityIndex] = distance
        map[neighborCityIndex][cityIndex] = distance

  // Find the shortest path using Dijkstra's algorithm
  For count from 0 to numberOfLocations - 2:
    min = infinity
    minIndex = -1

    // Find the next closest unvisited city
    For location from 0 to length of shortestDistances - 1:
      If not visited[location] and shortestDistances[location] <= min:
        min = shortestDistances[location]
        minIndex = location
    visited[minIndex] = true

    // If the destination city is reached, stop the algorithm
    If minIndex is equal to indexFinder(cities, destinationCity):
      break

    // Update shortest distance and shortest path cities
    For neighborIndex from 0 to numberOfLocations - 1:
      If shortestDistances[minIndex] + map[minIndex][neighborIndex] < shortestDistances[neighborIndex]
          and not visited[neighborIndex] and map[minIndex][neighborIndex] != 0
          and shortestDistances[minIndex] != maxVal:
        shortestDistances[neighborIndex] = shortestDistances[minIndex] + map[minIndex][neighborIndex]
        shortestPathCities[neighborIndex] = minIndex

  // Reconstruct the shortest path
  shortestPath = ArrayList of City objects

  // If there is no optimal path, specify that there's no path
  If shortestDistances[indexFinder(cities, destinationCity)] is equal to Double.MAX_VALUE:
    Print "No path could be found."
    Return shortestPath

  // Starting from the destination city, construct the shortest path backwards
  shortestPathCity = indexFinder(cities, destinationCity)
  While shortestPathCity is not -1:
    Add cities[shortestPathCity] to shortestPath
    shortestPathCity = shortestPathCities[shortestPathCity]

  // Reverse the shortest path
  start = 0
  end = size of shortestPath - 1
  While start < end:
    Swap shortestPath[start] with shortestPath[end]
    start++
    end--

  // Print the total distance and path
  Print "Total Distance: " + shortestDistances[indexFinder(cities, destinationCity)] + ". Path: "
  i1 = 0
  i2 = size of shortestPath - 1
  While i1 < i2:
    Print shortestPath[i1].cityName + " -> "
    i1++
  Print shortestPath[last element].cityName

  Return shortestPath
```



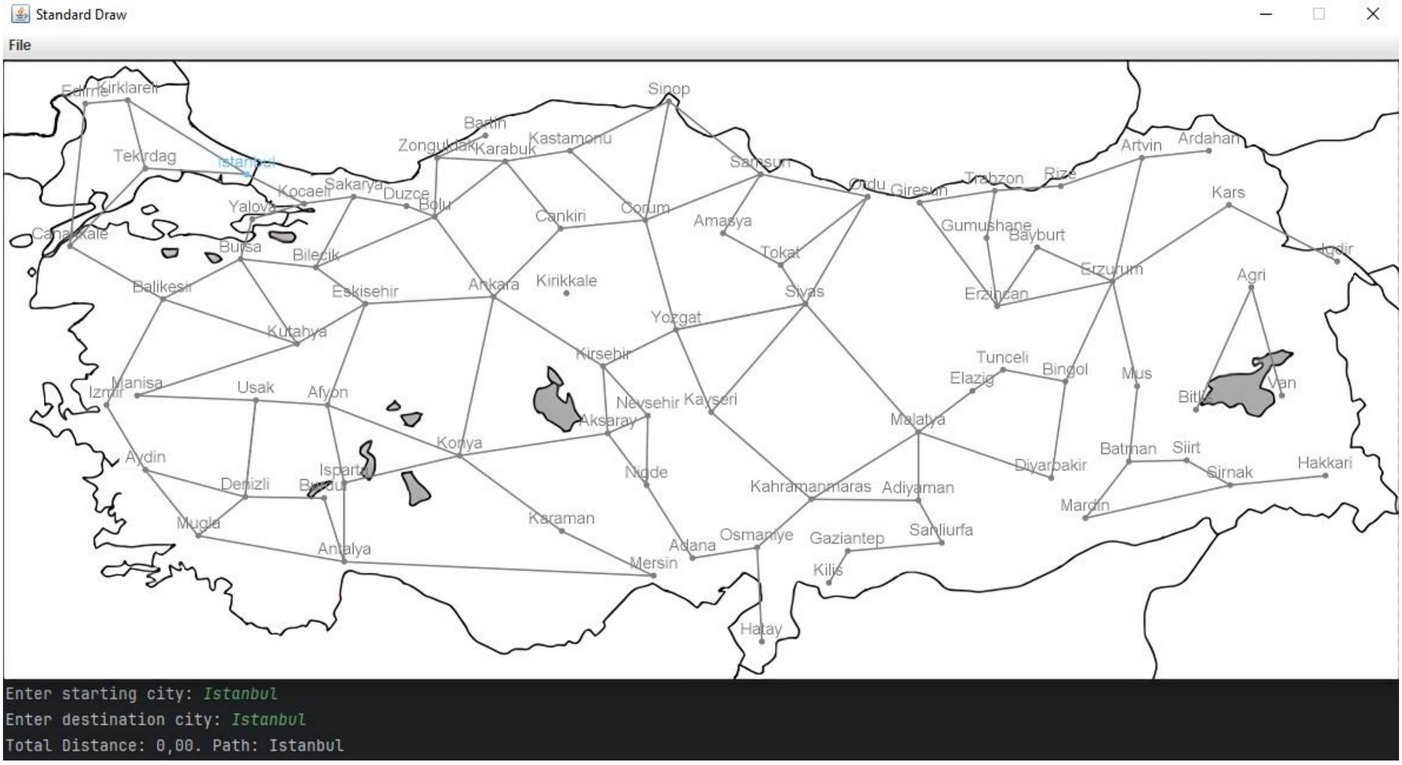
```
Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=U
Enter starting city: Edirne
Enter destination city: Giresun
Total Distance: 2585,49. Path: Edirne -> Kirklareli -> Istanbul -> Kocaeli -> Sakarya -> Duzce -> Bolu -> Ankara -> Kirsehir -> Yozgat -> Sivas -> Malatya -> Elazig -> Tunceli -> Bingol -> Erzurum -> Erzincan -> Giresun
```



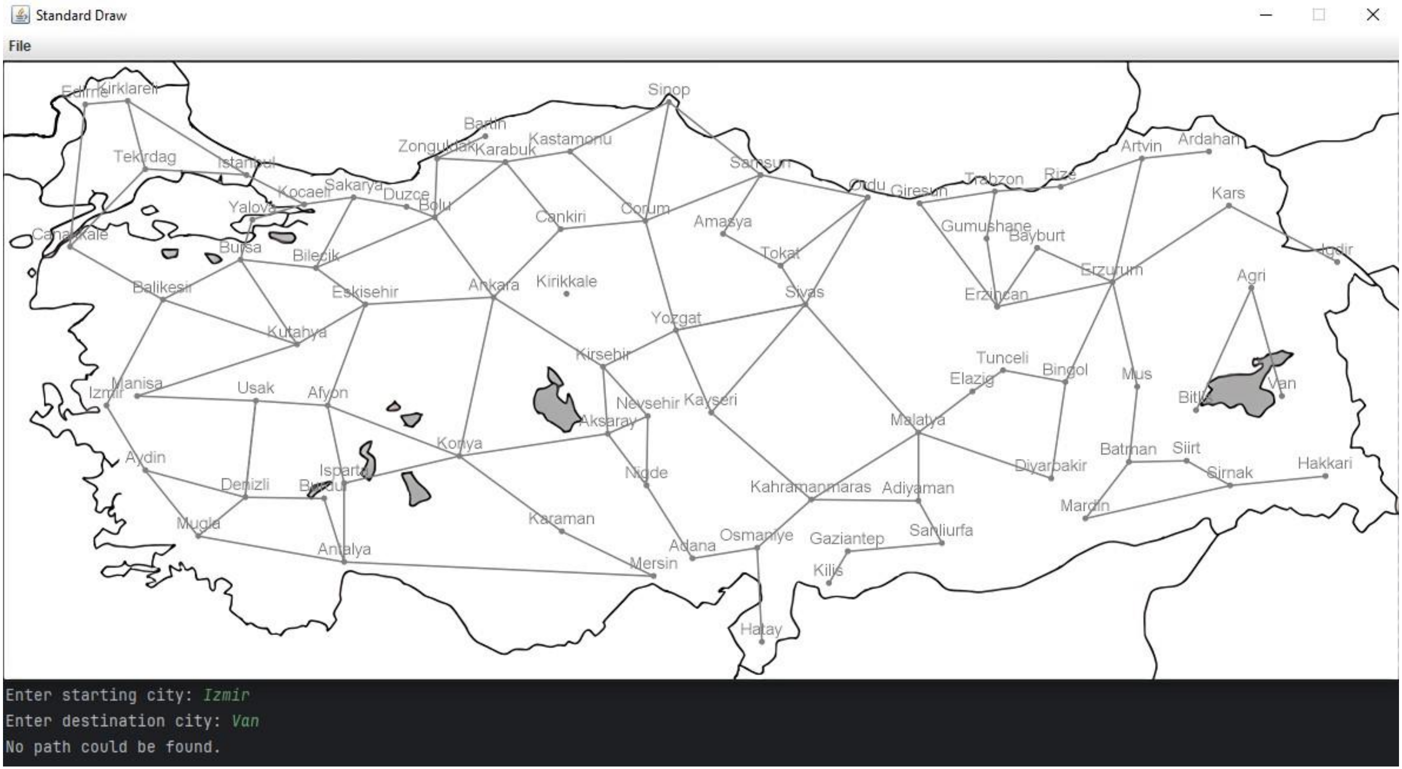
```
Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=
Enter starting city: Canakkale
Enter destination city: Hakkari
Total Distance: 2780,87. Path: Canakkale -> Balikesir -> Kutahya -> Eskisehir -> Ankara -> Kirsehir -> Yozgat -> Sivas -> Malatya -> Elazig -> Tunceli -> Bingol -> Erzurum -> Mus -> Batman -> Siirt -> Sirnak -> Hakkari
```



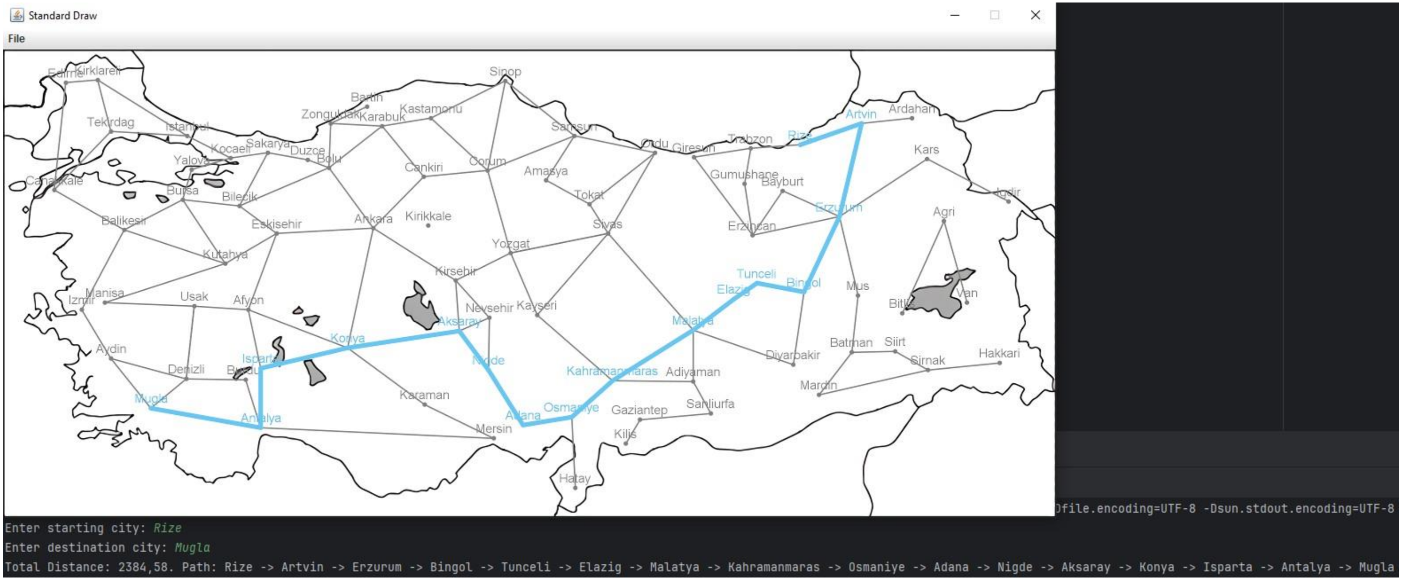
```
Enter starting city: Anka
City named 'Anka' not found. Please enter a valid city name.
Enter starting city: Ankara
Enter destination city: Deni
City named 'Deni' not found. Please enter a valid city name.
Enter destination city: Denizli
Total Distance: 689,10. Path: Ankara -> Eskisehir -> Afyon -> Usak -> Denizli
```



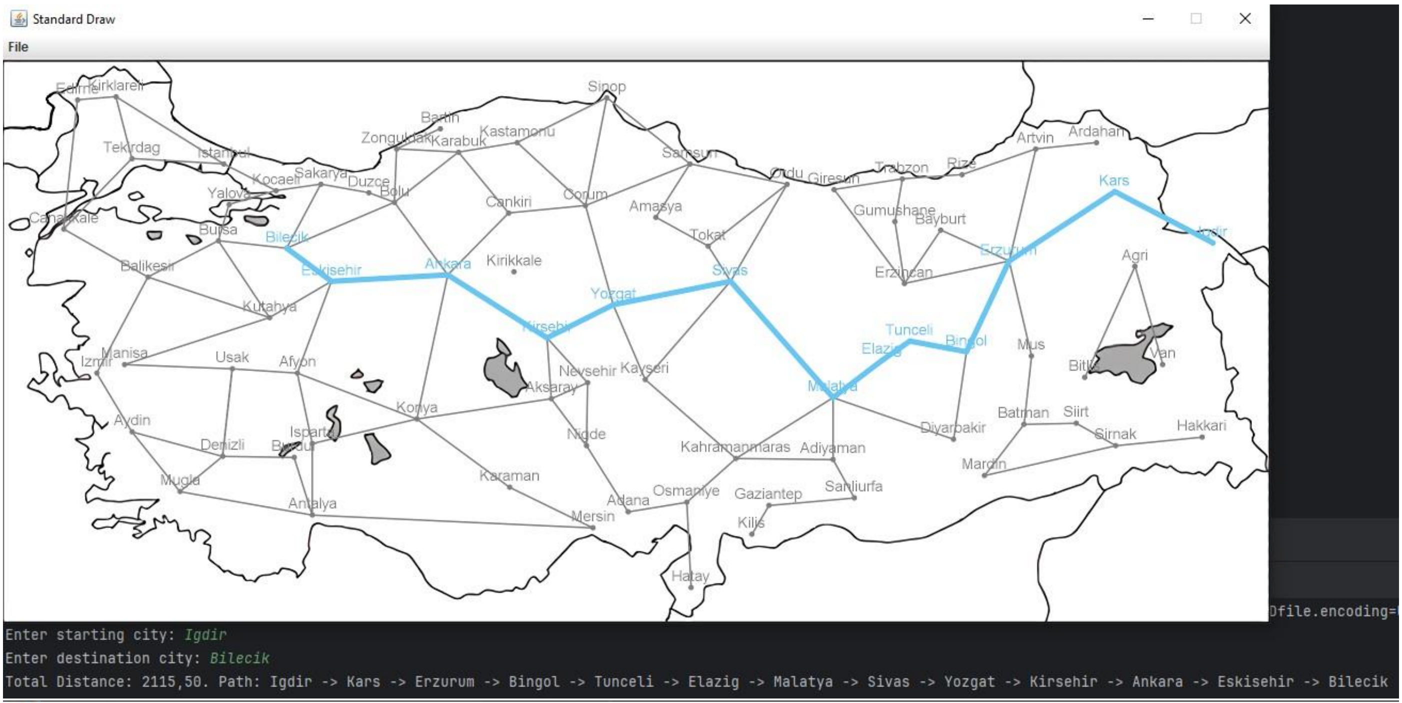
```
Enter starting city: Istanbul
Enter destination city: Istanbul
Total Distance: 0,00. Path: Istanbul
```



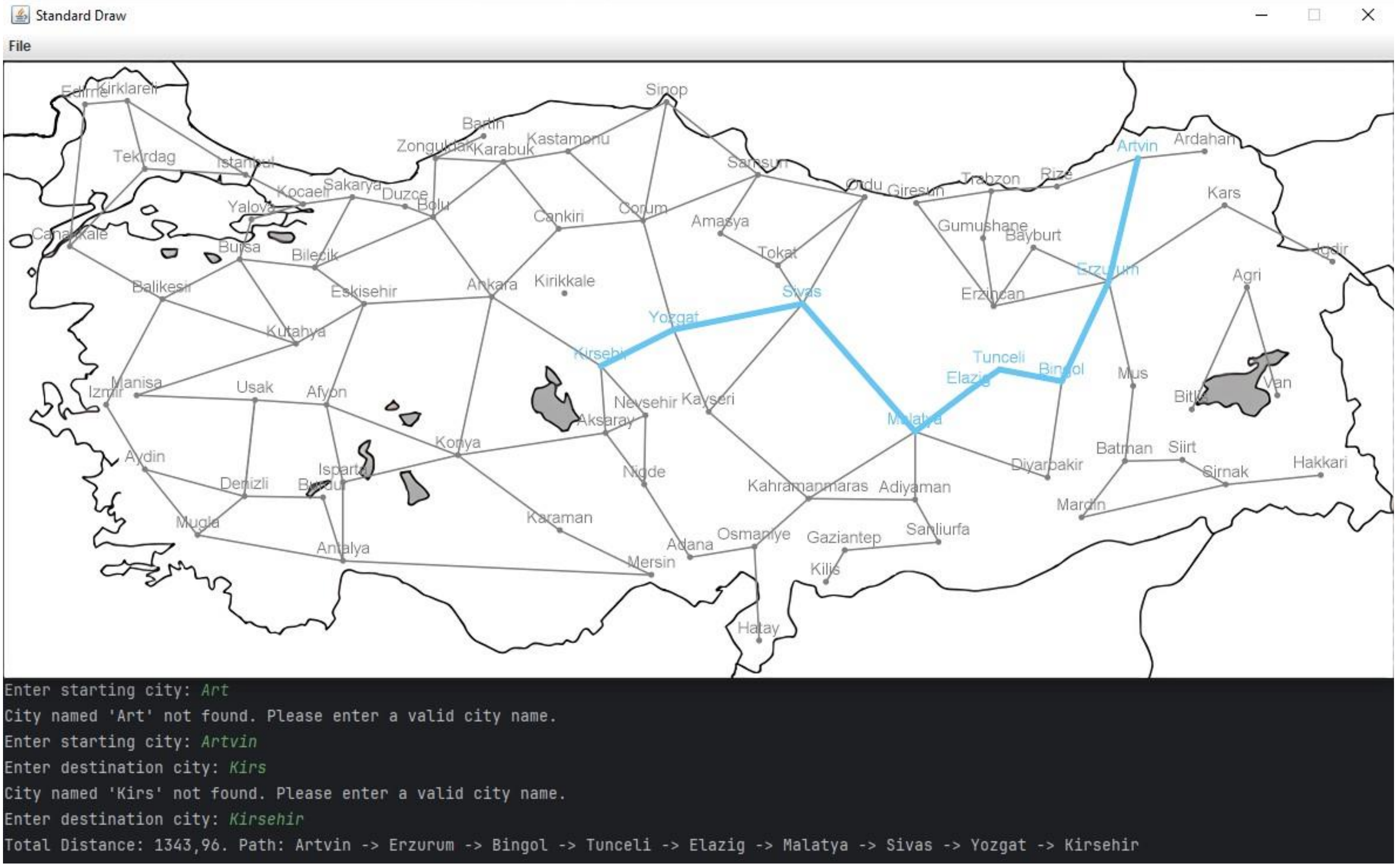
```
Enter starting city: Izmir
Enter destination city: Van
No path could be found.
```



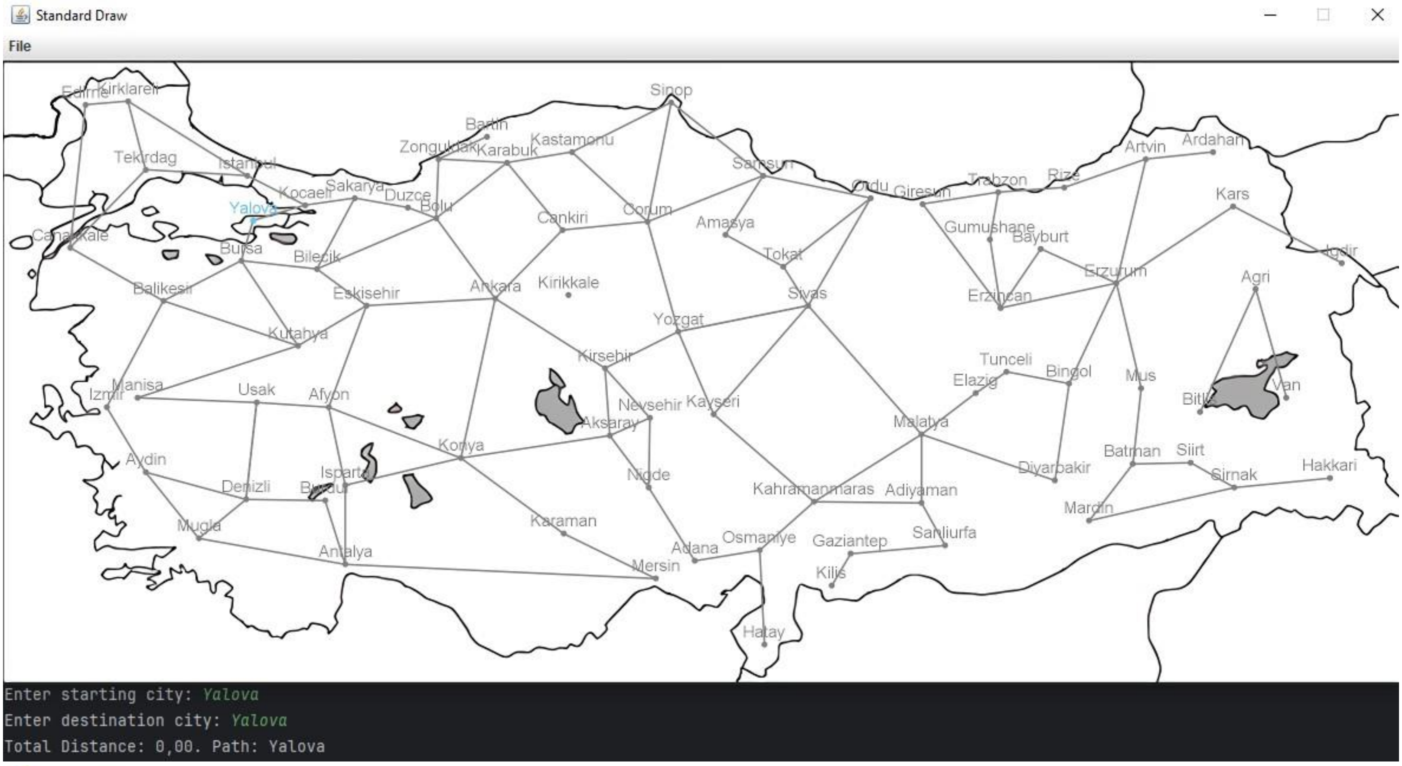
```
Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8
Enter starting city: Rize
Enter destination city: Mugla
Total Distance: 2384,58. Path: Rize -> Artvin -> Erzurum -> Bingol -> Tunceli -> Elazig -> Malatya -> Kahramanmaras -> Osmaniye -> Adana -> Nigde -> Aksaray -> Konya -> Isparta -> Antalya -> Mugla
```



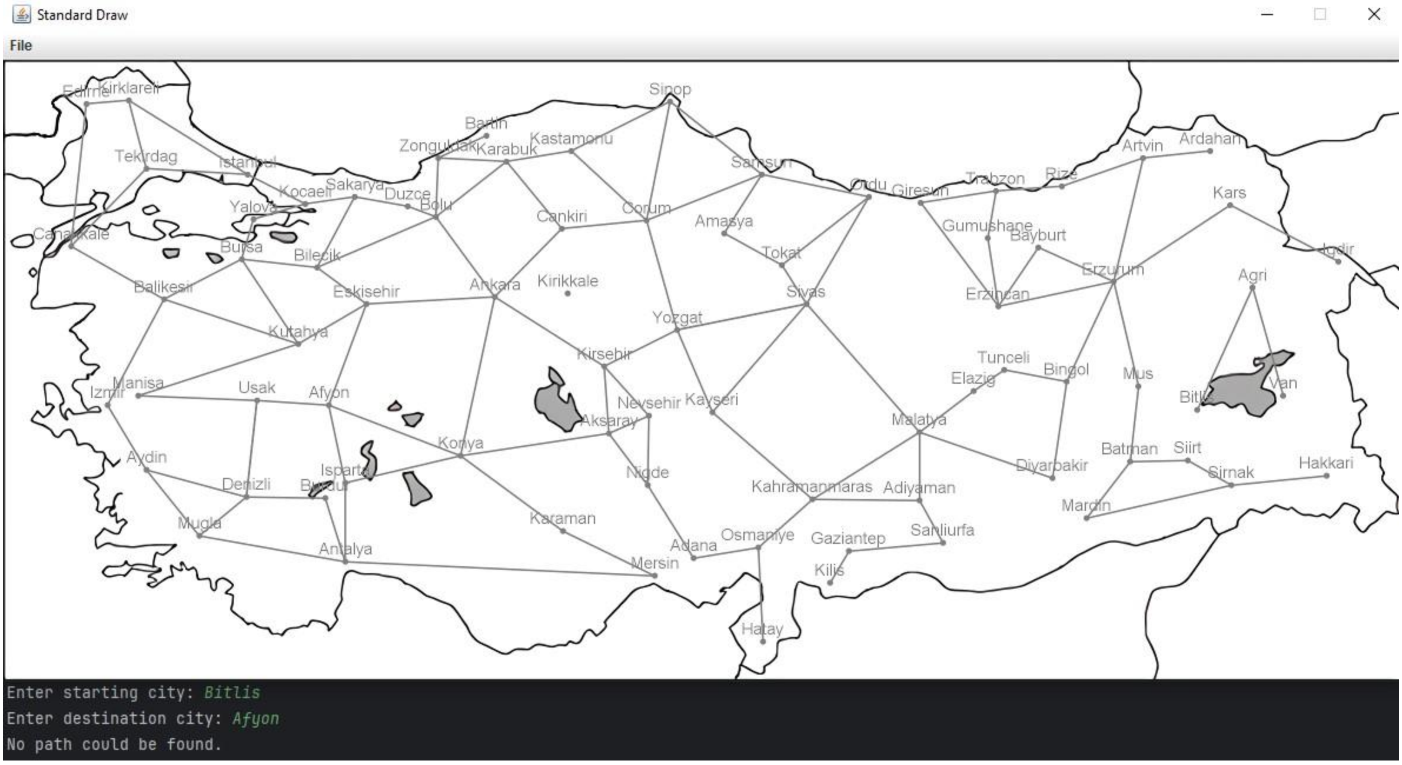
```
Enter starting city: Igdir
Enter destination city: Bilecik
Total Distance: 2115,50. Path: Igdir -> Kars -> Erzurum -> Bingol -> Tunceli -> Elazig -> Malatya -> Sivas -> Yozgat -> Kirsehir -> Ankara -> Eskisehir -> Bilecik
```



```
Enter starting city: Art
City named 'Art' not found. Please enter a valid city name.
Enter starting city: Artvin
Enter destination city: Kirs
City named 'Kirs' not found. Please enter a valid city name.
Enter destination city: Kirsehir
Total Distance: 1343,96. Path: Artvin -> Erzurum -> Bingol -> Tunceli -> Elazig -> Malatya -> Sivas -> Yozgat -> Kirsehir
```



```
Enter starting city: Yalova
Enter destination city: Yalova
Total Distance: 0,00. Path: Yalova
```



```
Enter starting city: Bitlis
Enter destination city: Afyon
No path could be found.
```