**The Shortest Path Problem Solution in Java**

The Shortest Path Program in Java

MEF University

Data Structures And Algorithms Course’s Final Project Report Paper

19.12.2024

**Instructor:**

Asst. Prof. Yassine DRIAS

**Authors:**

Mustafa GARİP, 042201173

Burak GÜVERÇİN, 042301112

Baran ÖZDEMİR, 042301077

Damla ŞENER, 042301130

Ege ÖZTÜRK, 042301101

Table Of Contents

[**1. Introduction 3**](#_lt4hzbz7c94o)

[**2. Implementation of Algorithms and Data Structures 3**](#_ahrufxe8gw7p)

[2.1. Custom Stack 3](#_aa1azojl79fe)

[2.2. Custom Queue 5](#_xtagsmo2sxve)

[2.3. Customized BFS Algorithm 7](#_dqtjqc2v7tjl)

[2.3.1. The BFS Algorithm 7](#_7jv341vq45ol)

[2.3.2. Customized BFS Algorithm 8](#_cvn57zu2t77t)

[2.3.3. The City Class 9](#_yle3hxwcog06)

[2.3.4. The Path Class 9](#_9m7l9wf3e41u)

[2.3.5. Detailed Explanation Of Customized BFS Algorithm 10](#_d4yojonp4y9p)

[2.4. Customized DFS Algorithm 12](#_mjcigvgznjwt)

[2.4.1. The DFS Algorithm 12](#_mvpx17orq85u)

[2.4.2. Customized DFS Algorithm 13](#_3rajeqwxaj81)

[2.4.3. Step-by-Step Explanation 14](#_m148lp9ulu73)

[**3. Implementation of The Graph and The Program 20**](#_mwidgaz3eu6c)

[3.1. The Graph 20](#_9chm7talg7mt)

[3.2. The Program 21](#_lnn6rsvlq6vv)

[3.2.1. Step-by-Step Explanation 23](#_e8rc1y3jsnic)

[**4. Complexity Analysis 25**](#_1vvd4t4ad5v6)

[4.1. Custom Stack 26](#_zcx8ei6e5rn8)

[4.2. Custom Queue 26](#_jbzfv9ccc3r9)

[4.3. Customized BFS Algorithm 27](#_dnjz2byeuqau)

[4.3.1. Theoretical Time Complexity 27](#_v90i1zgypo4)

[4.3.2. Empirical Time Complexity 28](#_j6b025u9z26c)

[4.4. Customized DFS Algorithm 28](#_roxk7f7c76iw)

[4.4.1. Theoretical Time Complexity 28](#_yo4angljga5a)

[4.4.2. Empirical Time Complexity 29](#_508emp4cm3wm)

[**5. Discussion About Limitations and Improvements 30**](#_8p4e7242c33v)

[5.1. Customized BFS Algorithm 30](#_gptvuwhsgauo)

[5.2. Customized DFS Algorithm 30](#_6yvhpgs4mmfz)

[**6. Results and Conclusion 31**](#_twwqaeraj3h2)

[6.1. Results 31](#_fpxzp6l1pd9a)

[6.2. Conclusion 32](#_nmzgp4r9ov9g)

# Introduction

For this project, our purpose was to develop a program that finds and calculates the shortest path between a starting city and an ending city via using data structures and algorithms. We tried to optimize the possible route finding processes which are being used in the fields like transportation and logistics. We used Custom Stack and Queue, versions of Depth-First Search (DFS), and Breadth-First Search (BFS) algorithms in order to reach our purpose, the optimum path-finding program. This report provides an overview of our work for this project with the parts of each team member; the coding and the explanation parts.

# Implementation of Algorithms and Data Structures

In this section, we will present the algorithms and data structures used in the project. Each will be explained individually, with every team member demonstrating their contributions to the algorithms.

## Custom Stack

The Custom Stack was created by Ege Öztürk.   
  
 The Custom Stack is designed without depending on Java’s existing classes. It is created with generic type <E> which allows the stack to store elements of any type. This provides type safety and reusability. Also an array of type E generic is used to store the elements in the stack. Our Custom Stack is created with many known methods such as push, pop, peek, isEmpty, size.   
  
Here is the breakdown of the methods:

Push: It allows us to add a new element to our Custom Stack. It also checks if the stack’s capacity is full and if the capacity is full, then the capacity is increased.

(This feature optimizes memory usage and allows adding unlimited elements)

Pop: It allows us to remove the top element of our Custom Stack and also returns that element.

Peek: It allows us to see the top element in our Custom Stack without removing it.

isEmpty: Checks if the Custom Stack is empty.

Size: Returns the current number of elements in our Custom Stack.

Here is the full code of Custom Stack with a detailed comment:

public class CustomStack<E> {

private E[] CustomStack; // This array will hold the elements of our stack

private int size; // This keeps track of how many elements are in the stack right now

private int maxCapacity; // The maximum number of elements our stack can hold

// Constructor to create a stack with a specific capacity

public CustomStack(int maxCapacity) {

this.maxCapacity = maxCapacity; // Set the max number of elements the stack can hold

size = 0; // Start with an empty stack

CustomStack = (E[]) new Object[maxCapacity]; // Create the array to hold the elements

}

// Default constructor to create a stack with a default capacity of 10

public CustomStack() {

maxCapacity = 10; // Default to a stack that can hold 10 elements

size = 0; // Start with an empty stack

CustomStack = (E[]) new Object[maxCapacity]; // Create the array to hold the elements

}

// Push a new element onto the stack

public void push(E element) {

// Check if the stack is full and needs more space

if (size == maxCapacity) {

int newCapacity = maxCapacity \* 2; // Double the size of the stack

E[] newStack = (E[]) new Object[newCapacity]; // Create a new bigger array

// Copy the old stack's elements to the new stack

for (int i = 0; i < size; i++) {

newStack[i] = CustomStack[i]; // Copy each element over

}

CustomStack = newStack; // Update our stack to be the new larger stack

maxCapacity = newCapacity; // Update the capacity to reflect the new size

}

// Add the new element to the top of the stack

CustomStack[size++] = element;

}

// Pop an element off the stack (remove it)

public E pop() {

if (isEmpty()) {

return null; // Return null because there's nothing to pop

}

// Get the element at the top of the stack and remove it

E element = CustomStack[--size]; // Decrease the size and get the top element

CustomStack[size] = null; // Clean up the old element

return element; // Return the popped element

}

// Peek at the top element of the stack without removing it

public E peek() {

if (isEmpty()) {

return null; // Return null because there's nothing at the top

}

// Return the top element without removing it

return CustomStack[size - 1];

}

// Check if the stack is empty

public boolean isEmpty() {

return size == 0; // If the size is 0, the stack is empty

}

// Get the current size of the stack (how many elements are in it)

public int size() {

return size; // Return the number of elements currently in the stack

}

}

## Custom Queue

The Custom Queue was created by Burak Güverçin. The Custom Queue is designed without depending on Java’s existing classes. It is created with generic type <E> which allows the queue to store elements of any type. This provides type safety and reusability. Also an array of type E generic is used to store the elements in the queue. Our Custom Queue is created with many known methods such as enqueue, dequeue, isEmpty, size. I also added a resize method to make the queue dynamic.   
  
Here is the breakdown of the methods:

enqueue(E element): It allows us to add a new element to our Custom Queue. It also checks if the stack’s capacity is full and if the capacity is full, then the capacity is increased.

(This feature optimizes memory usage and allows adding unlimited elements)

dequeue(): It allows us to remove the front element of our Custom Queue and also returns that element.

isEmpty(): Checks if the Custom Queue is empty.

resize(): Doubles the size when the queue is full.

Here is the full code of Custom Queue:

public class CustomQueue<E> {

// This array holds the elements of the queue

private E[] queue;

// The capacity of the queue

private int capacity;

// The index of the front and rear of the queue

private int front;

private int rear;

// The current number of elements in the queue

private int size;

/\*

Constructor to initialize the queue with a specified capacity.

(initialCapacity) The initial capacity of the queue.

\*/

public CustomQueue(int initialCapacity) {

this.capacity = initialCapacity;

this.queue = (E[]) new Object[capacity]; // Create an array to hold the queue's elements

this.size = 0; // Number of elements currently in the queue

this.front = 0; // Points to the front of the queue

this.rear = -1; // Points to the end of the queue (empty at first)

}

/\*

Adds a new element to the back of the queue.

If the queue is full, it makes more space by doubling the size.

(Element) The new element to add to the queue.

\*/

public void enqueue(E element) {

if (size == capacity) { // Check if the queue is full

resize();

}

rear = (rear + 1) % capacity; // Move the rear to the next position

queue[rear] = element; // Add the new element to the rear of the queue

size++; // Increase the size of the queue

if (size > 100000) { // Limit the queue to avoid OutOfMemoryError

dequeue();

}

}

/\*

Removes and returns the front element of the queue.

If the queue is empty, it returns null.

Return The dequeued element, or null if the queue is empty.

\*/

public E dequeue() {

if (isEmpty()) return null; // If no elements to dequeue, return null

E dequeuedElement = queue[front]; // Get the element at the front

queue[front] = null; // Clear the reference for garbage collection

front = (front + 1) % capacity; // Circularly move the front index

size--; // Reduce the size of the queue

return dequeuedElement; // Return the dequeued element

}

/\*

Checks if the queue is empty.

Return True if the queue is empty, false otherwise.

\*/

public boolean isEmpty() {

return size == 0; // If size is 0, then the queue is empty

}

/\*

Doubles the size of the queue when it becomes full.

\*/

private void resize() {

E[] newQueue = (E[]) new Object[capacity \* 2]; // Create a new array with double capacity

for (int i = 0; i < size; i++) { // Copy all elements to the new array

newQueue[i] = queue[(front + i) % capacity];

}

queue = newQueue; // Update the queue reference to the new array

front = 0; // Reset front to the start of the array

rear = size - 1; // Set rear to the last valid index

capacity \*= 2; // Double the capacity

}

}

## Customized BFS Algorithm

The customized Breadth-First Search algorithm was created and was implemented by Baran ÖZDEMİR. The BFS algorithm has 2 queues in its standard implementation. Hence, I used 2 queues. One queue is called open and the other queue is closed. The open queue stores incomplete partial paths and distances while the closed queue stores complete paths and the final distances. In order to store the partial paths, I needed a way to create the partial paths and the best way to create a path was by defining a class named “Path”. Inside the path class, I was storing the cities that I visited in that path in order. The class also had an int variable called “distance” storing the total distance taken for that path so that I can compare the distances and find the shortest path. I am storing the cities visited in the path object so I also have the “City” class. The Classes will be explained but first, let’s see the algorithm. Below are the standard and the modified BFS algorithms.

### The BFS Algorithm

**The Standard Algorithm BFS  
Input**: *s*: initial state; *F*: set of final states;  
**Output**: a solution (chain) if success, failure otherwise;  
**Var**: Open, Closed: initially empty queues;

**Begin**

1. insert the initial state *s* in *Open*;
2. if (*Open* = ∅) then **failure** else continue;
3. dequeue *n*, the first node in *Open* and insert it in *Closed*;
4. if *n* doesn't have successors then go to 2;
5. determine the successors of *n* and insert them in *Open*;
6. create a chain between these nodes and *n*;
7. if there is a final state among the successors then **success**: the solution is the chain of nodes going from the current node to the root;  
   else go to 2;

**End**

### Customized BFS Algorithm

**Algorithm:** Modified BFS

**Input:** s: initial state; f: final state;

**Output:** set of states with shortest distance;

**Var:** Open, Closed: initially empty queues;

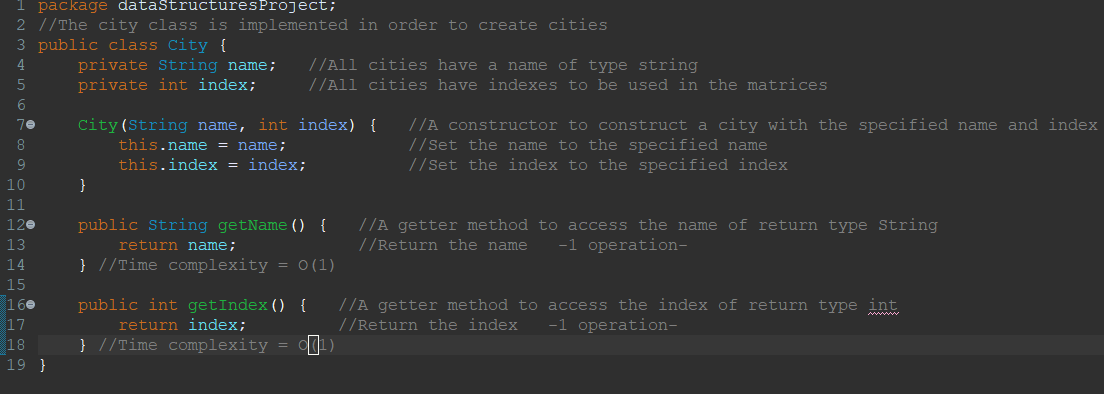
**Begin:**

1. Declare the initial path and add the first city into it.
2. Insert it in Open.
3. If Open is not Empty
4. Dequeue Open and assign it to the current path
5. Set the current city as the last city in the current path.
6. If the current city is the final/target city, Insert the current path in the Closed queue.
7. Else, for every neighborhood of the current city, do the following(8, 9, 10):
8. If the neighbor is not the same as the current or is not already in the path.
9. Create a new path consisting of the current path and the neighbor.
10. Enqueue the new path into Open.
11. If Closed is not Empty, continue with 12, else go to 17.
12. Declare a temporary path, dequeue Closed and set temporary as dequeued path.
13. If the total distance of the path is shorter than the current shortest distance
14. Set shortest distance as the temporary path’s distance
15. Set the shortest path as the temporary path.
16. Go back to 11.
17. Return the shortest path.

**End**

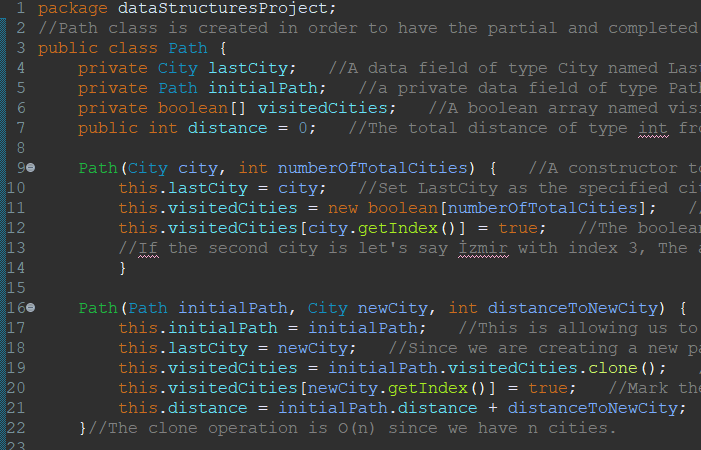
Before talking about the BFS algorithm in detail, I must represent the Path and City Classes.

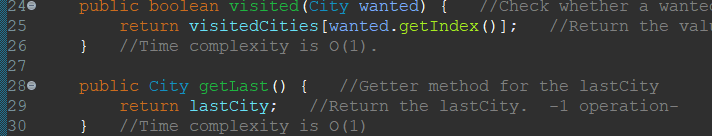
### The City Class



All cities have their name and index as private values. The name of the cities will be taken as input of type String so I needed a data field like that. The index will be used to find the distance in the distances array. Each city will have their index separately. Constructor will create a city with the given index and name. Since the name and indexes are private, getter methods are implemented.

### The Path Class





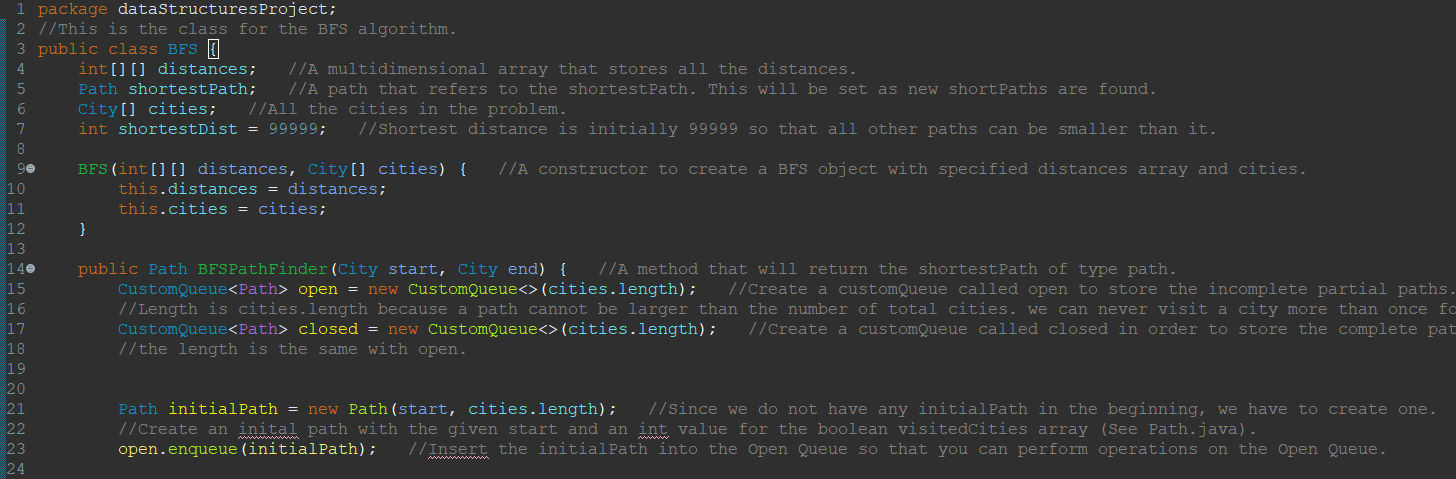
The path class was implemented by Baran ÖZDEMİR, Ege ÖZTÜRK and Burak GÜVERÇİN. The reason why we did it together was because of the Java Heap Space error. We tried to fix it together and we did it.

The Path class represents a sequence of cities and supports efficient operations for pathfinding algorithms. It tracks the last city in the path, the total distance traveled, and visited cities through a boolean array. The initialPath field keeps the initial parts of a path, reducing cloning operations when creating paths.

The class offers two constructors. The first initializes a path with a single city and marks it as visited. The second extends an existing path by adding a new city, updating the visited array, and increasing the total distance. Checking whether a city is visited has a time complexity of O(1) due to direct index access in the boolean array. Creating or extending paths also has a time complexity of O(1).

Overall, the Path class is optimized for efficient path management, with all primary operations having a time complexity of O(1).

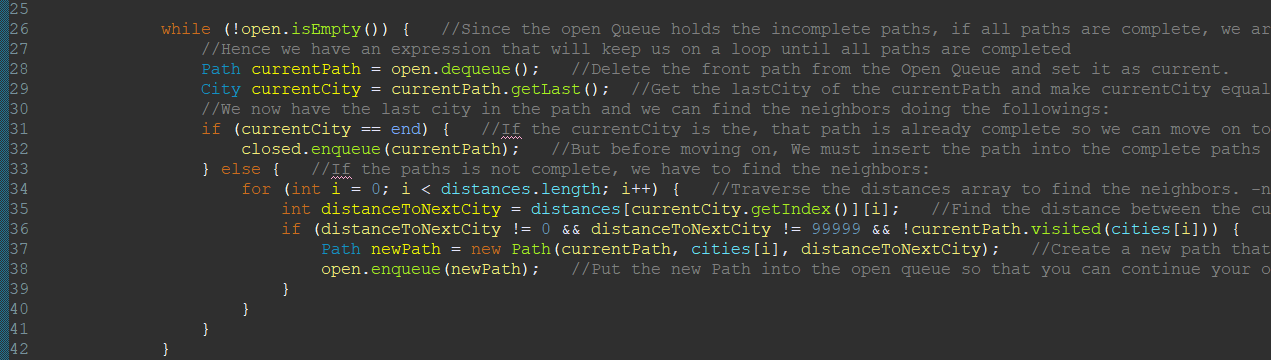
### Detailed Explanation of Customized BFS Algorithm

****

The BFS class is implemented in order to maintain the algorithm more efficiently. There are data fields: distances: int[][], shortestPath: Path, cities: City[] and shortestDist: int. Distances is the graph that we will be working on. It will store the distances between all cities. shortestPath will be our return value. We are trying to find the shortest path. After finding the shortest path, we have to store it. shortestPath is where we will store it. Cities is an array that stores all cities in an increasing order of index. We will use it to find the neighborhood with the help of the distances array. shortestDist is the variable that will hold the shortest value. It is set to 99999 so that all possible first paths can be the new shortest path.

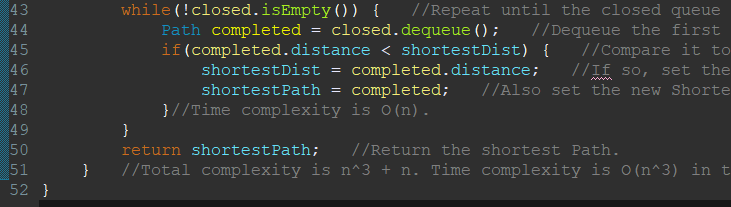
We then have the BFS constructor to create BFS objects with the given matrix and cities. After creating an object, we can call the BFSPathFinder method to find and return the shortestPath. The algorithm will start with the declaration of 2 data structures. Custom queues. One is called open and the other is closed.

Since we do not have any path but only the starting city, we have to declare and initialize our first path. This path will be used to create new paths.



After creating our initialPath and enqueueing it into the open queue. We can now search for the neighbors level by level. At the beginning, the open is NOT empty because there is the initialPath in it. So we go in the while loop and create a path called currentPath to keep track of our current path that we are working with. We will then dequeue the open set current path as the dequeued path. Now we have a path but we need its last city in order to find the neighbors. We use the getLast method and get the last city, declare a variable named currentCity to keep track of the city we are currently working with. So we now have the city. We then check if the city is our target. If so, we will enqueue the currentPath into closed and we will be done with it. Else, we need to find the neighbors so we go down into the for loop.

We traverse the distances array to find the distances. We create a variable named distanceToNextCity that stores the distance between the currentCity and its neighbor. If the distance is 0, it is the same city; if the distance is 99999, there is no road between them and if the neighbor is already in the path, it is already visited. If all of these conditions are wrong at the same time, then we have found a neighbor. We can now create a new path from the currentPath and the neighbor. We also need to increase the distance. This is done by the constructor. Then, we add the newPath into the open. We continue one by one for all of the first paths inside the open queue hence we are traversing the queue level by level because lower levels are enqueued and dequeued first. Once all paths are completed, they will be inserted into the closed queue, the open will be empty and we will go out of the loop.



Since the open queue is totally empty and all paths are complete and stored in the closed queue, we will now try to find the shortest path inside the closed queue. We can access the paths in the closed queue only by dequeuing them. Hence, we need to dequeue the element one by one and then the queue will be empty. When it is empty, we know that we are done. This is why we start with a while loop. We then dequeue the first path and store the dequeued path in a temporary variable called completed. If we do not store it in a temporary variable, it will be gone forever and we won’t be able to do a comparison. Then we are comparing the total distance of the completed path to the shortestDist. If smaller, it is the new shortest distance hence we set the shortestDist as completed distance and shortestPath as completed. We repeat the process and when we are out of the loop, we simply return the shortest path.

## Customized DFS Algorithm

The customized Depth-First Search algorithm was created and was implemented by Mustafa GARİP. First, In this part, we will first explain what the stock DFS algorithm is, then we see the demonstration of the pseudo code of the customized DFS algorithm, followed by a step-by-step breakdown of the customized DFS algorithm.

### The DFS Algorithm

**Input**: *s*: initial state; *F*: set of final states;  
**Output**: a solution (chain) if success, failure otherwise;  
**Var**: Open: an initially empty stack; Closed: an initially empty queue;

**Begin**

* push *s* into *Open*;
* if (*Open* = ∅) then **failure** else continue;
* pop *n*, the first node in *Open* and insert it in *Closed*;
* if *n* doesn't have successors or the depth of the tree is equal to the *threshold* then go to 2;
* determine the successors of *n* and push them into *Open*;
* create a chain between these nodes and *n*;
* if there is a final state among the successors then **success**: the solution is the chain of nodes going from the current node to the root;  
  else go to 2;
* **End**

The algorithm starts with an initial state s, and pushes it into a stack called Open. If the stack is empty, the search fails. The algorithm then repeatedly pops the top node n from the Open, and marks it as visited by throwing it into Closed, and checks if it has successors. If n has no successors or a threshold is reached, the algorithm goes back to process the next node. Otherwise, the successors of n are determined and pushed into Open, while a connection is created between n and its successors. If any of these successors is a final state, the algorithm succeeds, and the solution is the path from the current node back to the root. If no final state is found, the algorithm continues until it finds a solution, and fails if no solution exists.

Asst. Prof. Yassine DRIAS gave us the algorithm's pseudo code, and I decided to make improvements and some modifications to implement the DFS that finds the shortest path between two nodes. You will be able to observe the algorithm below (see 2.4.2).

### Customized DFS Algorithm

**Input**: *graph*: graph of cities; current*City*: initial city, node; *destination*; desired city, node; *shortestPath*: set of cities, final path;  
**Output**: a solution (*shortestPath*) if success, failure otherwise;  
**Var**: *Stack*: an initially empty stack;

**Begin**

* push *currentCity* into *Stack*;
* while (*Stack* != ∅) then continue else go to 9;
* pop *currentCity*, the first node in *Stack*;
* if *currentCity* isthe *destination,* then look at possible paths, (if exists) go backwards on the *graph* and decide the *shortestPath* (chain of cities), then go to 2;
* If *currentCity* is not the *destination* then determine the successors of *currentCity* and push them into *Stack*;
* If *currentCity* is not the *destination* and *currentCity* doesn't have successors then go to 2;
* If premise 2 doesn’t meet, then return *shortestPath,* **success**;
* If premise 2 doesn’t meet, and (shortestPath == null) then **failure**;
* **End**

First, I drew a blueprint of the custom algorithm with the graph on a piece of paper, I tried to find the shortest path like the DFS Algorithm, and I designed my custom DFS algorithm.

The algorithm starts from the currentCity, it pushes cities into the stack. If it finds the destination node, it goes backwards through the graph to determine the shortest path and returns it. If a city isn’t the destination, its neighbors are added to the stack and the algorithm continues the exploration. In the end, the algorithm checks if a path was found. If it is, it returns success with the shortestPath; else, it returns failure.

The biggest difference between two algorithms is that the customized algorithm does not use visited queue logic, unlike the stock algorithm. I will talk about this problem at (see 5.4).

In the following part, I will explain my algorithm that was implemented as a code in Java step-by-step in full detail.

### Step-by-Step Explanation

Here’s the step-by-step explanation of the customized DFS that used on the program for finding the shortest path between two cities:

public class DFS

{

private int[][] adjacencyMatrix;

private String[] cities;

private int numberOfCities;

public DFS(int[][] adjacencyMatrix, String[] cities)

{

this.adjacencyMatrix = adjacencyMatrix;

this.cities = cities;

this.numberOfCities = cities.length;

}

public String[] dfs(String from, String to)

{

CustomStack<Integer> stack = new CustomStack<>();

String[] shortestPath = new String[numberOfCities];

int shortestPathLength = 0;

int bestKnownDistance = Integer.MAX\_VALUE;

int[] parent = new int[numberOfCities];

int[] distances = new int[numberOfCities];

for (int i = 0; i < numberOfCities; i++)

{

parent[i] = -1;

distances[i] = Integer.MAX\_VALUE;

}

int firstCity = findCityIndex(from);

int lastCity = findCityIndex(to);

distances[firstCity] = 0;

stack.push(firstCity);

while (!stack.isEmpty())

{

int currentCity = stack.pop();

if (currentCity == lastCity)

{

String[] currentPath = new String[numberOfCities];

int elevator = currentCity;

int currentPathLength = 0;

while (elevator != -1)

{

currentPath[currentPathLength] = cities[elevator];

currentPathLength++;

elevator = parent[elevator];

}

if (currentPath.length > 1)

reverseArray(currentPath, currentPathLength);

int currentPathDistance = calculatePathDistance(currentPath, currentPathLength);

if (shortestPath[0] == null || currentPathDistance < bestKnownDistance)

{

for (int i = 0; i < currentPathLength; i++)

shortestPath[i] = currentPath[i];

shortestPathLength = currentPathLength;

bestKnownDistance = currentPathDistance;

}

continue;

}

for (int neighbor = 0; neighbor < numberOfCities; neighbor++)

{

int distanceBetween = adjacencyMatrix[currentCity][neighbor];

if (distanceBetween != 99999 && distanceBetween > 0)

{

int newDistance = distances[currentCity] + distanceBetween;

if (newDistance < distances[neighbor])

{

distances[neighbor] = newDistance;

parent[neighbor] = currentCity;

stack.push(neighbor);

}

}

}

}

return trimArray(shortestPath, shortestPathLength);

}

private void reverseArray(String[] array, int length)

private String[] trimArray(String[] array, int length)

public int findCityIndex(String city)

private String normalize(String input)

public int calculatePathDistance(String[] path)

public int calculatePathDistance(String[] path, int length)

}

**First block:**

private int[][] adjacencyMatrix;

private String[] cities;

private int numberOfCities;

We have 3 variables in the DFS class;

* **adjacencyMatrix:** The adjacency matrix of the graph.
* **cities (array):** String array that stores all cities names.
* **numberOfCities:** Length of the cities array.

**The constructor of the class:**

public DFS(int[][] adjacencyMatrix, String[] cities)

{

this.adjacencyMatrix = adjacencyMatrix;

this.cities = cities;

this.numberOfCities = cities.length;

}

**The String[] dfs(from, to) method and its variables:**

public String[] dfs(String from, String to)

{

CustomStack<Integer> stack = new CustomStack<>();

CustomQueue<Integer> visited = new CustomQueue<>(numberOfCities);

String[] shortestPath = new String[numberOfCities];

int shortestPathLength = 0;

int bestKnownDistance = Integer.MAX\_VALUE;

int[] parent = new int[numberOfCities];

int[] distances = new int[numberOfCities];

for (int i = 0; i < numberOfCities; i++)

{

parent[i] = -1;

distances[i] = Integer.MAX\_VALUE;

}

int firstCity = findCityIndex(from);

int lastCity = findCityIndex(to);

The dfs method takes “from” and “to” String inputs and sets the shortest path between them, as a String array, a chain of cities.

The variables of dfs(from, to) method with short explanations;

* **stack:** Custom stack that keeps cities.
* **shortestPath:** A String array that keeps the chain of cities.
* **shortestPathLength:** shortestPath’s length, we use this to recreating shortestPath.
* **bestKnownDistance**: The shortest path’s length (IRL).We will use this to compare the shortestPath and currentPath.
* **parent:** An int array that provides tracking the graph, we use this to go backward of a path. There isn’t a parent for the cities, so all parent indexes are -1.
* **distances:** An int array that stores distances for all cities. There is no distance defined initially, so all distances are infinite.

**The DFS Algorithm:**

distances[firstCity] = 0;

stack.push(firstCity);

while (!stack.isEmpty())

{

int currentCity = stack.pop();

if (visited.contains(currentCity))

continue;

The algorithm starts with assigning distances of the first city’s index zero and pushes it into the stack. Then, DFS starts to search. The current city pops from the stack.

if (currentCity == lastCity)

{

String[] currentPath = new String[numberOfCities];

int elevator = currentCity;

int currentPathLength = 0;

while (elevator != -1)

{

currentPath[currentPathLength] = cities[elevator];

currentPathLength++;

elevator = parent[elevator];

}

if (currentPath.length > 1)

reverseArray(currentPath, currentPathLength);

int currentPathDistance = calculatePathDistance(currentPath, currentPathLength);

if (shortestPath[0] == null || currentPathDistance < bestKnownDistance)

{

for (int i = 0; i < currentPathLength; i++)

shortestPath[i] = currentPath[i];

shortestPathLength = currentPathLength;

bestKnownDistance = currentPathDistance;

}

continue;

}

This code block checks if the current city is our destination, if it is, the while loop and variable elevator try to access the root, and this loop creates currentPath String array that records the all cities between the root city and the current city through the parent array. If we reverse the currentPath array, we get the path that the DFS explored that is wanted by us. In the end, the if block asks whether that is the shortest path we ever found, if so, record into shortestPath and the path’s length, bestKnownDistance.

for (int neighbor = 0; neighbor < numberOfCities; neighbor++)

{

int distanceBetween = adjacencyMatrix[currentCity][neighbor];

if (distanceBetween != 99999 && distanceBetween > 0)

{

int newDistance = distances[currentCity] + distanceBetween;

if (newDistance < distances[neighbor])

{

distances[neighbor] = newDistance;

parent[neighbor] = currentCity;

stack.push(neighbor);

}

}

}

}

In this code block, the for-loop tries to access the current city’s neighbors to find a path. The distance between two neighbors is obtained from adjacencyMatrix. If a neighbor exists, the distance is recorded and the algorithm connects the network between the neighbors, and the neighbor is pushed into the stack and joins the algorithm’s loop.

return trimArray(shortestPath, shortestPathLength);

}

}

In the end of the algorithm, the shortest path is returned by the algorithm, the solution no matter what. If we were to work with big numbers and datas, I would never return the shortest path like this. However, time is a valuable resource. I will mention this topic on (see 5.4).

**Miscellaneous\*:**

private void reverseArray(String[] array, int length)

**reverseArray(String[], int):** Reverses an array. (O(n))

private String[] trimArray(String[] array, int length)

**trimArray(String[], int):** Trims an array, we use this to trim the shortestPath array because

shortestPath.length = numberOfCities at the start, we must redesign the array. (O(n))

public int findCityIndex(String city)

**findCityIndex(String):** Finds a city’s index, the algorithm processes cities with ints. (O(n))

private String normalize(String input)

**normalize(String):** Helps to pick up the Turkish characters like ş,ç,ö,ü. (O(1))

public int calculatePathDistance(String[] path)

**calculatePathDistance(String[]):** Calculates the path according to input “path”. (O(n))

private int calculatePathDistance(String[] path, int length)

**calculatePathDistance(String[], int):** Calculates the path according to inputs “path” and the

path’s length “length”. (O(n))

**Note\*:** The n that is in Big-O notation implies the number of cities. FYI.

# Implementation of The Graph and The Program

## The Graph

Here is the graph, int[][] type matrix named adjacencyMatrix:

int[][] adjacencyMatrix =

{

// İstanbul Ankara İzmir Bursa Adana Gaziantep Konya Diyarbakır Antalya Mersin Kayseri Şanlıurfa Malatya Samsun Denizli Batman Trabzon

{0, 449, 99999, 153, 99999, 99999, 645, 99999, 690, 956, 776, 99999, 99999, 737, 649, 99999, 99999},

// Ankara

{449, 0, 591, 389, 484, 705, 266, 1003, 483, 501, 317, 848, 682, 402, 483, 99999, 732},

// İzmir

{99999, 591, 0, 333, 898, 1118, 560, 99999, 451, 911, 874, 99999, 99999, 1003, 238, 99999, 99999},

// Bursa

{153, 389, 333, 0, 856, 1075, 507, 99999, 546, 869, 715, 1212, 1049, 750, 480, 99999, 1091},

// Adana

{99999, 484, 898, 856, 0, 225, 346, 525, 649, 95, 307, 369, 389, 719, 758, 618, 851},

// Gaziantep

{99999, 705, 1118, 1075, 225, 0, 568, 315, 785, 311, 357, 151, 251, 803, 974, 409, 838},

// Konya

{645, 266, 560, 507, 346, 568, 0, 866, 303, 360, 306, 702, 729, 663, 386, 964, 896},

//Diyarbakır

{99999, 1003, 99999, 99999, 525, 315, 866, 0, 99999, 610, 571, 182, 235, 803, 1276, 97, 586},

// Antalya

{690, 483, 451, 546, 649, 785, 303, 99999, 0, 631, 610, 924, 99999, 99999, 217, 99999, 99999},

// Mersin

{956, 501, 911, 869, 95, 311, 360, 610, 631, 0, 99999, 99999, 99999, 99999, 99999, 99999, 99999},

// Kayseri

{776, 317, 874, 715, 307, 357, 306, 571, 610, 99999, 0, 99999, 99999, 99999, 99999, 99999, 99999},

// Şanlıurfa

{99999, 848, 99999, 1212, 369, 151, 702, 182, 924, 99999, 99999, 0, 99999, 99999, 99999, 99999, 99999},

// Malatya

{99999, 682, 99999, 1049, 389, 251, 729, 235, 99999, 99999, 99999, 99999, 0, 99999, 99999, 99999, 99999},

// Samsun

{737, 402, 1003, 750, 719, 803, 663, 803, 99999, 99999, 99999, 99999, 99999, 0, 99999, 858, 99999},

// Denizli

{649, 483, 238, 480, 758, 974, 386, 1276, 217, 99999, 99999, 99999, 99999, 99999, 0, 99999, 99999},

// Batman

{99999, 99999, 99999, 99999, 618, 409, 964, 97, 99999, 99999, 99999, 99999, 99999, 858, 99999, 0, 99999},

// Trabzon

{99999, 732, 99999, 1091, 851, 838, 896, 586, 99999, 99999, 99999, 99999, 99999, 99999, 99999, 99999, 0}

};

We got the adjacency matrix from a .csv file that Asst. Prof. Yassine DRIAS provided us for the program. This .csv file includes 17 cities and their distances between each other. Then, we created an int[][] variable named “adjacencyMatrix” and filled it with all rows and columns with cities. By this way, we created the graph abstractly, as a matrix. Both of the algorithms get the adjacency matrix as input and handle separately their shortest paths. We believe that this way is the simplest way to create a city graph or tree for the program.

## The Program

Here is the shortest path finder program(All members of the group participated in this part):

import java.util.Arrays;

import java.util.Scanner;

public class Program

{

public static void main(String[] args)

{

City istanbul = new City("İstanbul", 0);

City ankara = new City("Ankara", 1);

City izmir = new City("İzmir", 2);

City bursa = new City("Bursa", 3);

City adana = new City("Adana", 4);

City gaziantep = new City("Gaziantep", 5);

City konya = new City("Konya", 6);

City diyarbakir = new City("Diyarbakır", 7);

City antalya = new City("Antalya", 8);

City mersin = new City("Mersin", 9);

City kayseri = new City("Kayseri", 10);

City sanliurfa = new City("Şanlıurfa", 11);

City malatya = new City("Malatya", 12);

City samsun = new City("Samsun", 13);

City denizli = new City("Denizli", 14);

City batman = new City("Batman", 15);

City trabzon = new City("Trabzon", 16);

City[] bfsCities = {istanbul, ankara, izmir, bursa, adana, gaziantep, konya, diyarbakir, antalya, mersin, kayseri, sanliurfa, malatya, samsun, denizli, batman, trabzon};

String[] dfsCities = {"İstanbul", "Ankara", "İzmir", "Bursa", "Adana", "Gaziantep", "Konya", "Diyarbakır", "Antalya", "Mersin", "Kayseri", "Şanlıurfa", "Malatya", "Samsun", "Denizli", "Batman", "Trabzon"};

int[][] adjacencyMatrix =

{

//We already explained, please look at beginning of the section 3.2

};

Scanner scanner = new Scanner(System.in);

DFS dfsAlgorithm = new DFS(adjacencyMatrix, dfsCities);

BFS bfsAlgorithm = new BFS(adjacencyMatrix, bfsCities);

System.out.println("Welcome to The Shortest Path Finder Program! This program helps you find the shortest path between two cities of your choice.\nHere's the cities:");

for (int i = 0; i < dfsCities.length; i++)

System.out.println(" \* " + dfsCities[i]);

System.out.println("Simply enter the starting city (from-city) and the destination city (to-city), and we'll calculate the shortest path for you.");

while (true)

{

try

{

System.out.println("What is your starting city?");

String to = scanner.nextLine();

System.out.println("What is your destination city?");

String from = scanner.nextLine();

System.out.println();

City toBFS = findCityByName(to, bfsCities);

City fromBFS = findCityByName(from, bfsCities);

long dfsStartTime = System.nanoTime();

String[] shortestPath = dfsAlgorithm.dfs(to, from);

long dfsEndTime = System.nanoTime();

System.out.println("The shortest path found using the DFS algorithm.");

System.out.println("Shortest Path, DFS: " + Arrays.toString(shortestPath));

System.out.println("Distance: " + dfsAlgorithm.calculatePathDistance(shortestPath) + "km");

System.out.println("DFS Execution Time: " + (dfsEndTime - dfsStartTime) / 1\_000\_000.0 + " milliseconds");

System.out.println();

long bfsStartTime = System.nanoTime();

Path myPath = bfsAlgorithm.BFSPathFinder(toBFS, fromBFS);

long bfsEndTime = System.nanoTime();

System.out.println("The shortest path found using the BFS algorithm.");

System.out.println(myPath.toString());

System.out.println("BFS Execution Time: " + (bfsEndTime - bfsStartTime) / 1\_000\_000.0 + " milliseconds");

System.out.println("\nThank you for using our program!\n");

break;

}

catch (CityNotFoundException e)

{

System.err.println(e.getMessage());

}

}

scanner.close();

}

public static City findCityByName(String cityName, City[] cities)

private static String normalize(String input)

}

**Appendix\*:** The program is a raw version without a UI; you can access the version with UI (ProgramUI.java) on Blackboard.

### Step-by-Step Explanation

First, we will give source codes, then we will explain them. Here is the step-by-step explanation of the raw program:

public class Program

{

public static void main(String[] args)

{

City istanbul = new City("İstanbul", 0);

City ankara = new City("Ankara", 1);

City izmir = new City("İzmir", 2);

City bursa = new City("Bursa", 3);

City adana = new City("Adana", 4);

City gaziantep = new City("Gaziantep", 5);

City konya = new City("Konya", 6);

City diyarbakir = new City("Diyarbakır", 7);

City antalya = new City("Antalya", 8);

City mersin = new City("Mersin", 9);

City kayseri = new City("Kayseri", 10);

City sanliurfa = new City("Şanlıurfa", 11);

City malatya = new City("Malatya", 12);

City samsun = new City("Samsun", 13);

City denizli = new City("Denizli", 14);

City batman = new City("Batman", 15);

City trabzon = new City("Trabzon", 16);

City[] bfsCities = {istanbul, ankara, izmir, bursa, adana, gaziantep, konya, diyarbakir, antalya, mersin, kayseri, sanliurfa, malatya, samsun, denizli, batman, trabzon};

String[] dfsCities = {"İstanbul", "Ankara", "İzmir", "Bursa", "Adana", "Gaziantep", "Konya", "Diyarbakır", "Antalya", "Mersin", "Kayseri", "Şanlıurfa", "Malatya", "Samsun", "Denizli", "Batman", "Trabzon"};

int[][] adjacencyMatrix;

In this part of the code, we assigned the cities for both algorithms. The BFS Algorithm gets the cities from a City array named bfsCities because the BFS handles the cities as City data type. However, the DFS Algorithm handles the cities as String data type, so dfsCities stores String unlike the BFS. And, the BFS will use bfsCities, which is City[], the DFS will use dfsCities, which is String[], to create a shortest path. This is caused by both algorithms approaching the path differently.

Scanner scanner = new Scanner(System.in);

DFS dfsAlgorithm = new DFS(adjacencyMatrix, dfsCities);

BFS bfsAlgorithm = new BFS(adjacencyMatrix, bfsCities);

System.out.println("Welcome to The Shortest Path Finder Program! This program helps you find the shortest path between two cities of your choice.\nHere's the cities:");

for (int i = 0; i < dfsCities.length; i++)

System.out.println(" \* " + dfsCities[i]);

System.out.println("Simply enter the starting city (from-city) and the destination city (to-city), and we'll calculate the shortest path for you.");

Basically, the salutation part of the program, but there is a point here. As you can see, we created DFS and BFS class instances and they received adjacencyMatrix and their cities from the Program class to run their algorithms.

while (true)

{

try

{

System.out.println("What is your starting city?");

String to = scanner.nextLine();

System.out.println("What is your destination city?");

String from = scanner.nextLine();

System.out.println();

City toBFS = findCityByName(to, bfsCities);

City fromBFS = findCityByName(from, bfsCities);

long dfsStartTime = System.nanoTime();

String[] shortestPath = dfsAlgorithm.dfs(to, from);

long dfsEndTime = System.nanoTime();

System.out.println("The shortest path found using the DFS algorithm.");

System.out.println("Shortest Path, DFS: " + Arrays.toString(shortestPath));

System.out.println("Distance: " + dfsAlgorithm.calculatePathDistance(shortestPath) + "km");

System.out.println("DFS Execution Time: " + (dfsEndTime - dfsStartTime) / 1\_000\_000.0 + " milliseconds");

System.out.println();

long bfsStartTime = System.nanoTime();

Path myPath = bfsAlgorithm.BFSPathFinder(toBFS, fromBFS);

long bfsEndTime = System.nanoTime();

System.out.println("The shortest path found using the BFS algorithm.");

System.out.println(myPath.toString());

System.out.println("BFS Execution Time: " + (bfsEndTime - bfsStartTime) / 1\_000\_000.0 + " milliseconds");

System.out.println("\nThank you for using our program!\n");

break;

}

catch (CityNotFoundException e)

{

System.err.println(e.getMessage());

}

}

scanner.close();

}

The while-loop provides continuity to program when an error occurs. So we have try-catch in this loop. Then, the program asks a user starting and destination city. If everything goes well, the user enters the start and destination correctly. The algorithms start to calculate the shortest path between cities entered by the user, and the results are printed with each algorithm’s execution times. If the user does not / can not enter the cities correctly, the program sends an exception, CityNotFoundException, and this error is transmitted to the user. The user enters the cities again thanks to the while-loop.

private String normalize(String input)

**normalize(String):** Helps to pick up the Turkish characters like ş,ç,ö,ü. (O(1))

City findCityByName(String cityName, City[] cities)

**findCityByName(String, City[]):** Finds a city (City Class) by its name (String Class), helps to

the implementation of the BFS algorithm. (O(n))

# Complexity Analysis

In this part, we will take over the empirical and theoretical time complexity of the codes. Each part of the program will be handled separately.

## Custom Stack

In this part, we analyze the time complexity of the Custom Stack and the time complexity of the CustomStack class is determined by the operations performed on the stack.

And here is the time complexity of the methods in the CustomStack class:

**push(E element)**

* Adding an element is an O(1) time operation unless the stack reaches the max capacity. If that happens, the array will be resized and the operation takes O(n) time because we need to copy all the elements to the new array.

**pop()**

* Removing an element from the top of the stack is an O(1) time which is a constant time operation.

**peek()**

* Accessing the top element of the stack is O(1) time which is a constant time operation.

**isEmpty()**

* Checking the variable “size” if it is zero or not zero is an O(1) time which is a constant time operation.

**size()**

* Returning the value of the variable “size” is a O(1) time which is a constant time operation.

## Custom Queue

In this part, we analyze the time complexity of the Custom Queue and the time complexity of the CustomQueue class is determined by the operations performed on the queue.

And here is the time complexity of the methods in the CustomStack class:

**enqueue(E element)**

* The resize operation takes **O(n)** time because we need to copy all the elements to the new array.
* Moving the rear pointer and adding the new element itself is **O(1)**.

**dequeue()**

* Removing an element from the front and updating the front pointer is **O(1)**.

**contains(E element)**

* The method checks each element of the queue from front to rear. In the worst case, it will traverse all n elements. Time complexity is **O(n)**.

**isEmpty()**

* This method simply checks if size == 0, which is a constant time operation. Time complexity is **O(1)**.

**resize()**

* This method checks if the specified element is in the queue. So it is going to check whether the element is in the queue or not. Due to usage of for loop the time complexity will be **O(n)**.

In conclusion, the overall time complexity will be **O(n)**.

## Customized BFS Algorithm

In this part, we analyze the time complexity of the customized BFS algorithm.

We will try to find the time complexity of the BFS algorithm but we already analysed the Path class.

### Theoretical Time Complexity

The theoretical time complexity will be determined by the number of operations and inputs. We will consider that the number of inputs is n. All of the operations done are 1 outside the algorithm(method) because they are declaring-initializing operations. Each is considered as 1 operation.

* The BFS class starts with 4 declarations. This is 4 operations. 4 is O(1)
* Constructor’s time complexity is O(1) because it has 2 operations inside. 2 is O(1).
* The pathFinderBFS method is complicated. It starts with 2 operations. 2 queue declarations. Then, 2 more operations, create a path and insert it in the open. 4 operations by far in the method. Then we have a while loop and a for loop inside it. We can start from the for loop and then go back to the while loop. Inside the for loop, we have cities[i], Path new Path = … and enqueue operations. Path class has a .clone operation and this is “n” operations. Since n is bigger, I will neglect the other operations inside the loop (Others are 1 operation). The for loop is done n times hence we have n^2 now. The while loop has 2 operations and an if statement. The if statement can not be executed every time hence the operations above it have higher time complexity. We can neglect the if statement to simplify the calculation. We have 2 operations n^2 times. This is 2 \* n^2. This is a loop hence it will become n \* n^2 = n^3
* The while loop to find the shortest path has a while loop. Inside the loop, there is an operation and an if statement. The if statement consists of 2 operations. The worst case is that the shortest distance is at the last part and is always decreasing. The if statement is 2 operations and the completed path is also 2 operations. The total operation is 4n.

n^3 + 4n is O(n^3). Hence, the time complexity is ***O(n^3)***

### Empirical Time Complexity

The empirical time complexity is calculated via milliseconds. The average empirical time complexity was hard to calculate since there are a lot of pairs of cities but I have found the average of 8 pair cities in the results part. The average of the 8 pair cities is approximately **41.53** milliseconds. Entering more than 17 will of course increase the time complexity and visa versa.

## Customized DFS Algorithm

In this part, we analyze the time complexity of the customized DFS algorithm. Theoretical complexity estimates are derived by investigating the structure of the algorithm, while empirical complexity is obtained through runtime measurements.

### Theoretical Time Complexity

The DFS algorithm processes each node in the graph to explore all possible paths. Here is the breakdown of the customized DFS algorithm’s time complexity:s

**Initialization, O(n)**

The parent and distances arrays are initialized, O(n).

**The DFS Traversal, O(n^2)**

Pushing & popping a city into the stack, O(1)

Finding neighbor loop, O(n^2), n cities, n neighbors in worst case.

**Path Construction, O(n)**

Reversing the path array, O(n).

Computing the total distance of the path, O(n)  
Comparing with the best-known path, O(n)

Hence, the overall theoretical time complexity of the algorithm is ***O(n^2)*** in the worst case\*.   
  
**Note\*:** If the graph is fully connected, the algorithms shall travel n cities and n neighbors.

**Note\*\*:** You can see detailed the mathematical theoretical time complexity of the algorithm on the “DFS” source code. FYI.

### Empirical Time Complexity

I measured the runtime of the algorithm to get the empirical time complexity of the algorithm. Each test was run 5 times, and the average runtime was recorded using System.nanoTime().

**Experiment:**

**Starting city:** İstanbul

**Destination city:** Samsun

**The Path:** İstanbul - 737km -> Samsun (Both cities are in all adjacency matrices (n = 8, 16, 17).)

Removed cities from the adjacencyMatrix:   
**n = 16:** Denizli.

**n = 8:** Denizli, Ankara, İzmir, Bursa, Adana, Gaziantep, Konya, Antalya.

| **Number of Cities (n)** | **Number of Paths (m)** | **Average Runtime (ms)** |
| --- | --- | --- |
| 8 | 11 | 0.08 |
| 16 | 79 | 0.15 |
| 17 | 88 | 1.02 |

**Experiment Results:**

The experiment shows that the algorithm works well for small graphs but gets slower as the graph becomes larger and especially has more paths. I want to draw attention that average runtimes are increasing polynomially, and it is in the theoretical time complexity (O(n^2)) borders. Although, the number of cities and the average runtimes of (n = 8 and n = 16) are almost proportional to 1, when the number of cities is 17, the runtime increases polynomial. This may be caused by deleting the complex nodes.   
  
**Note by Mustafa Garip:** Actually, I have been wanting so much to do an ultra comprehensive experiment about the time complexity of the algorithm, I was not able to do that due to project and final exams week, every seconds count. I know that the experiment is not enough to talk about the complexity of the algorithm. However, I gave my best for the project and the report. I wanted the paper to be written as a real academic report, actually it is.

# Discussion About Limitations and Improvements

We successfully created the shortest path program. However, each member of the team faced mistakes, bugs and logic errors. However, we beat all kinds of problems. In this part, we will discuss what we did face, and what we could do to improve and optimize the program.

## Customized BFS Algorithm

The BFS algorithm in the beginning faced the terrifying Java Heap Space error. In the beginning, the problem of cloning occurred. Since the BFS algorithm creates new paths each time a new path is found, the old version was cloning the Path which was implemented using an array of cities. The approach is changed into a reversed linked list approach. It is also similar to the parent child approach but the parent here is the initial path. Memory has been a problem throughout the BFS algorithm and Path class. Also, The BFS algorithm is slower compared to the DFS algorithm because the time complexity is higher.

One improvement would be to find a better approach for the visited cities array. Cloning is taking some really good time and is using a lot of memory.

## Customized DFS Algorithm

**Problems & Challenges:** When I started to script the algorithm, I thought the visited system should be in the algorithm. I designed my algorithm with visited, However, the visited system blocked finding the shortest path. Then I realize that we try to get the shortest path, we do not need to check whether the cities are visited or not. We only calculate and compare the path possibilities. I decided to remove the visited system due to a mistake that I made during designing the algorithm.

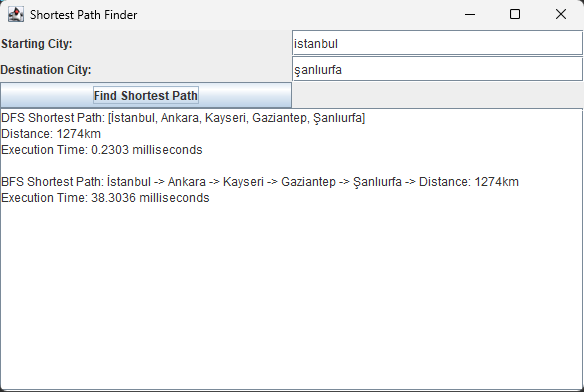
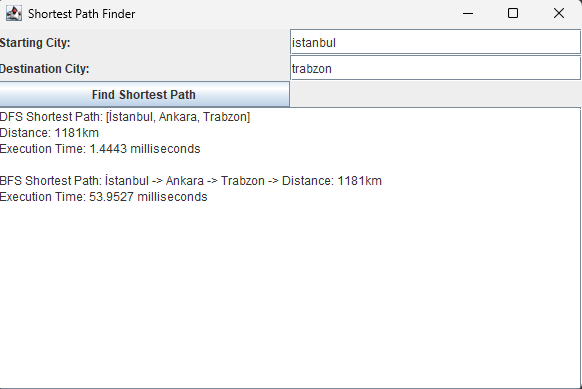
Also, the DFS backtracking system really made me overthink. However, I implemented –simple and reliable– elevator, parent & distances system that can record every step on the graph. The algorithm successfully backtracked when it arrived at the destination city through the system.

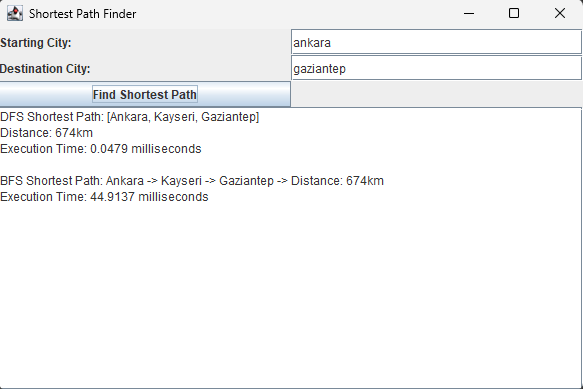
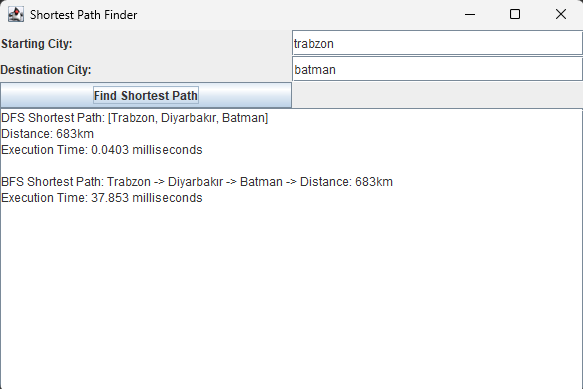
**Improvements:** I believe that the DFS’s last (return shortestPath) part would be improved. Only returning the shortest path is not healthy and not reliable. Defining a failure case would be healthy for the algorithm. Although I did not apply it, The program works without any problem, thanks to the graph.

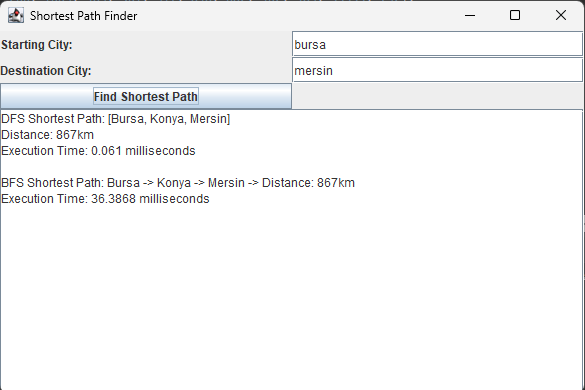
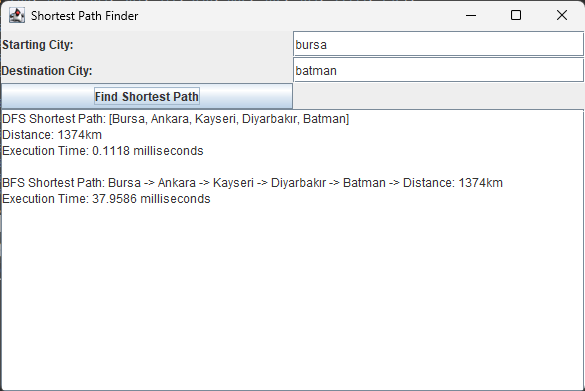
# Results and Conclusion

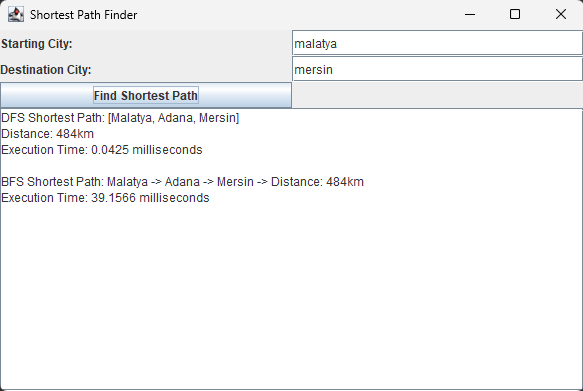
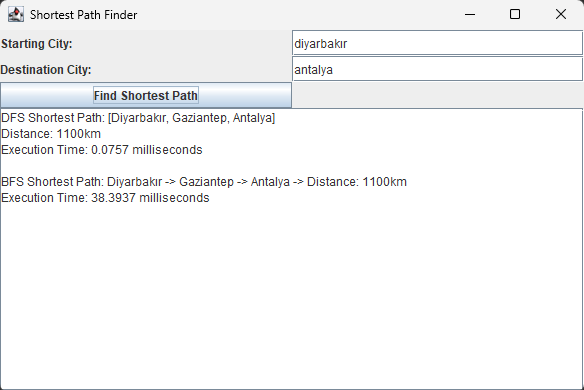
## Results

We run the program to observe 8 pairs of cities, and their shortest path between them. Here’s the achieved results:









In conclusion, we implemented our program using the javax library. As in the example we got exactly the same numbers that we should have. We used DFS and BFS algorithms to take these outcomes. In the DFS algorithm we used the CustomStack class which we created ourselves without using any imports. In the BFS algorithm we used the CustomQueue class which we created ourselves without using any imports. And we generated the main program to make algorithms run. And we created the UI of the program using the javax library. Finally, we tested the program by entering the city names and we got the correct results.

## Conclusion

In this project, we successfully developed a program which is finding and calculating the shortest path between a starting city and an ending city using the data structures and algorithms that we learned in the lectures. By implementing the custom versions of both stack and queue, we successfully applied DFS and BFS algorithms to find the shortest path between a starting city and an ending city. Through this project, we learned how to use algorithms and data structures to solve real-world problems. We saw how important it is to optimize solutions for better results. This project showed us how algorithms can help solve complex problems.