12.01.2024

MEF UNIVERSITY

YOUR FREEDOM IN LEARNING

COMP 201
DATA STRUCTURES AND ALGORITHMS
SHORTEST PATH PROBLEM

Hüseyin Karaman 042201080

Sinem Özbey 042101122

Muhammed Baha Karadağlı 042101106

Semih Yılmaz 042101116

TABLE OF CONTENTS

Abstract

This research optimizes Turkish city routes, delving into paths using depth-first graph traversal (DFS) and navigating systematically with breadth-first graph traversal (BFS). We explore nuanced differences in their effectiveness, performance and space complexities across diverse network configurations. We highlight alternatives like Dijkstra, Bellman-Ford and A* algorithms would be more proper fit for this task regarding their efficiency for intricate networks. Also, this paper sheds light on the limitations and possible future enhancements of these algorithms. Extracting insights from the Turkish Cities.csv file, our approach offers optimal solutions for the Shortest Path challenge by utilizing BFS and DFS algorithms.

1. Introduction

Efficiently determining the shortest path within intricate networks poses a persistent challenge. Our project addresses this by utilizing Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms, providing tailored solutions for user-preferred routes. Through a user-friendly Java-based interface, users can input starting and destination points, selecting between DFS and BFS options. The system seamlessly imports relevant data, enabling efficient exploration. Our primary goal is to assist users in navigating between specific points by developing intuitive algorithms, focusing on user-friendly design, and ensuring data security through file manipulation. This project not only tackles the challenge of finding optimal routes but also contributes to a deeper understanding of algorithmic performances in real-world scenarios, enhancing our knowledge of computational problem-solving challenges and their practical implications.

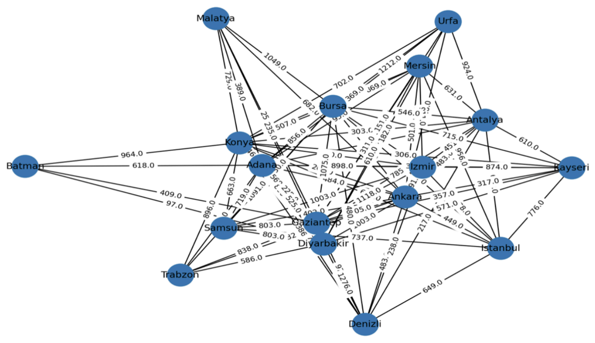


Figure 1. Network Graph of City-to-City Distances in Turkey

2. Traversal Algorithms for Graphs within Data Structure

- **Graph:** In computer science, a graph is a data structure that consists of nodes (or vertices) and edges. These nodes are interconnected, and the edges represent relationships or connections between the nodes.
- **Traversal Algorithms:** Traversal refers to the process of visiting and exploring each node or vertex in a data structure. In the context of graphs, traversal algorithms define how we move through the nodes of the graph.
- **Graph Traversal Algorithms:** These are specific methods or techniques designed to navigate through the nodes of a graph systematically. The two primary graph traversal algorithms are Depth-First Search (DFS) and Breadth-First Search (BFS).

- **Data Structure:** A data structure is a way of organizing and storing data to perform operations efficiently. In the case of graph traversal algorithms, the graph itself serves as the data structure.

- Putting it all together, "Traversal Algorithms for Graphs within Data Structure" refers to the techniques used for systematically exploring and navigating through nodes in a graph, where the graph itself is the underlying data structure. Depth-First Search and Breadth-First Search are common algorithms employed for this purpose, each with its distinct approach to visiting nodes in the graph.

- **Breadth-First Search (BFS):** The Breadth-First Search (BFS) algorithm serves as a tool to identify the shortest path between two points. In its essence, BFS mirrors the methodical exploration process, akin to navigating a city by first checking nearby locations before extending the search [1]. The algorithm systematically examines neighboring areas, gradually expanding its exploration to find the most direct route. Think of it as navigating a maze, exploring adjacent paths step by step to efficiently discover the shortest route. The process begins by designating a starting point as "visited" and employs a queue data structure following the First In First Out (FIFO) principle [1]. The systematic approach involves dequeuing the front node, examining and enqueuing unvisited neighboring nodes until the queue is empty, ensuring a comprehensive and efficient exploration of the graph.

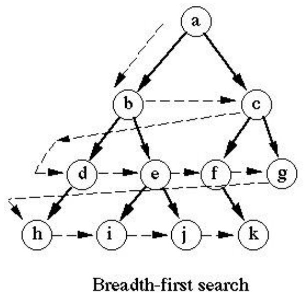


Fig 2. Breadth First Search [7]

- **Depth-First Search (DFS):** Depth-First Search (DFS) is a fundamental algorithm employed for finding the shortest path between two points, such as cities. Its strategy involves thoroughly exploring one path before considering alternatives, allowing DFS to delve deeply into a route before backtracking, similar to navigating dense areas and retracing steps at a dead-end. In practical applications, DFS efficiently explores various paths to determine the shortest distance between points [2]. In traversing data structures like trees and graphs, DFS starts at a chosen root node, systematically exploring each branch entirely before backtracking. Operating in a downward motion, DFS employs a stack following the Last In First Out (LIFO) principle to recall the next vertex when encountering a dead-end. This method, beneficial for tasks like path determination and identifying connected components, ensures comprehensive exploration of each branch before concluding when the stack is empty, indicating the visitation of every reachable node [2].

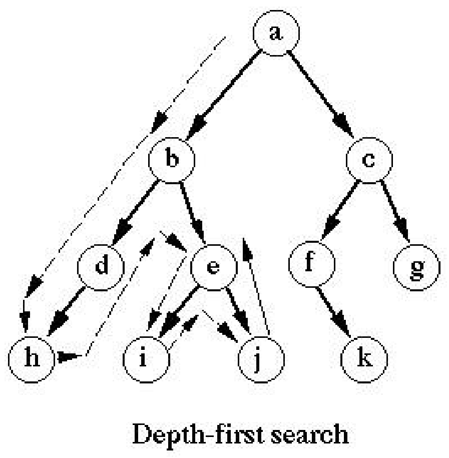


Fig 3. Depth First Search [7]

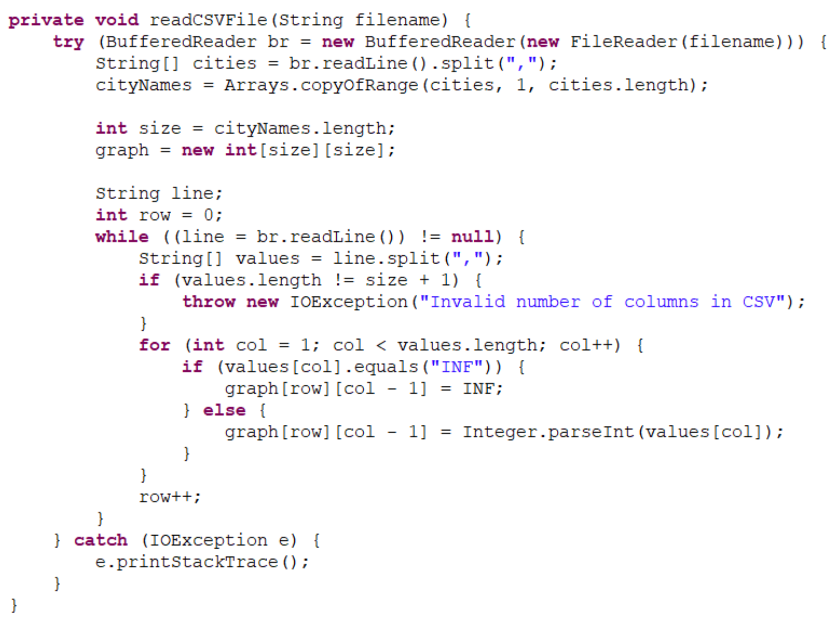
3. IMPLEMENTATION DETAILS OF THE JAVA PROGRAM

Executive Summary:

Our Java program employs the Swing library to create a user-friendly interface. The main menu offers options to find the shortest distance using DFS or BFS. Within the DFS and BFS modules, users can input starting and destination cities. The program seamlessly integrates with a graph data structure, optimizing pathfinding. The Graph.java reads data from the "TurkishCities.csv" file and constructs a stack. In BFS.java and DFS.java, these data structures obtained from Graph.java are utilized with their respective algorithms to find the shortest path. Main.java calls these functions, executing the project and prompting users, through the designed interface, to find the shortest path between their desired cities.

1. GRAPH CLASS:

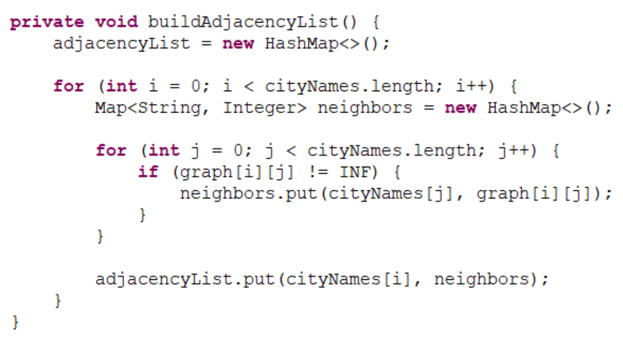
1.1. readCSVFile Function: The "readCSVFile" function reads city information from a CSV file. It extracts city names from the first line and initializes a 2D array called "graph" based on the number of cities. It then fills the array with distances between cities, handling special cases like "INF" for infinity. If there's an issue reading the file, it prints an error message.



```
private void readCSVFile(String filename) {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String[] cities = br.readLine().split(",");
        cityNames = Arrays.copyOfRange(cities, 1, cities.length);

        int size = cityNames.length;
        graph = new int[size][size];

        String line;
        int row = 0;
        while ((line = br.readLine()) != null) {
            String[] values = line.split(",");
            if (values.length != size + 1) {
                throw new IOException("Invalid number of columns in CSV");
            }
            for (int col = 1; col < values.length; col++) {
                if (values[col].equals("INF")) {
                    graph[row][col - 1] = INF;
                } else {
                    graph[row][col - 1] = Integer.parseInt(values[col]);
                }
            }
            row++;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 4. ReadCSVFile Function

1.2. buildAdjacencyList Function: The "buildAdjacencyList" function creates an adjacency list for cities. It uses a HashMap called "adjacencyList" to store each city and its neighbors with corresponding distances. The function iterates through the "graph" array, adding non-infinite distances to the neighbors' map. Finally, it adds the city and its neighbors to the main adjacency list.

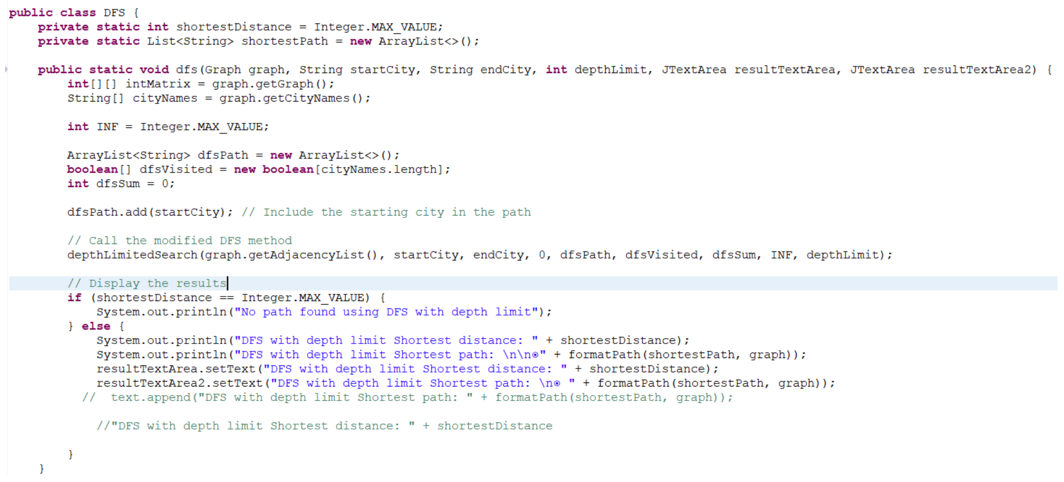


```
private void buildAdjacencyList() {
    adjacencyList = new HashMap<>();

    for (int i = 0; i < cityNames.length; i++) {
        Map<String, Integer> neighbors = new HashMap<>();

        for (int j = 0; j < cityNames.length; j++) {
            if (graph[i][j] != INF) {
                neighbors.put(cityNames[j], graph[i][j]);
            }
        }

        adjacencyList.put(cityNames[i], neighbors);
    }
}
```

Fig 5. buildAdjacency Function

2. DFS CLASS:

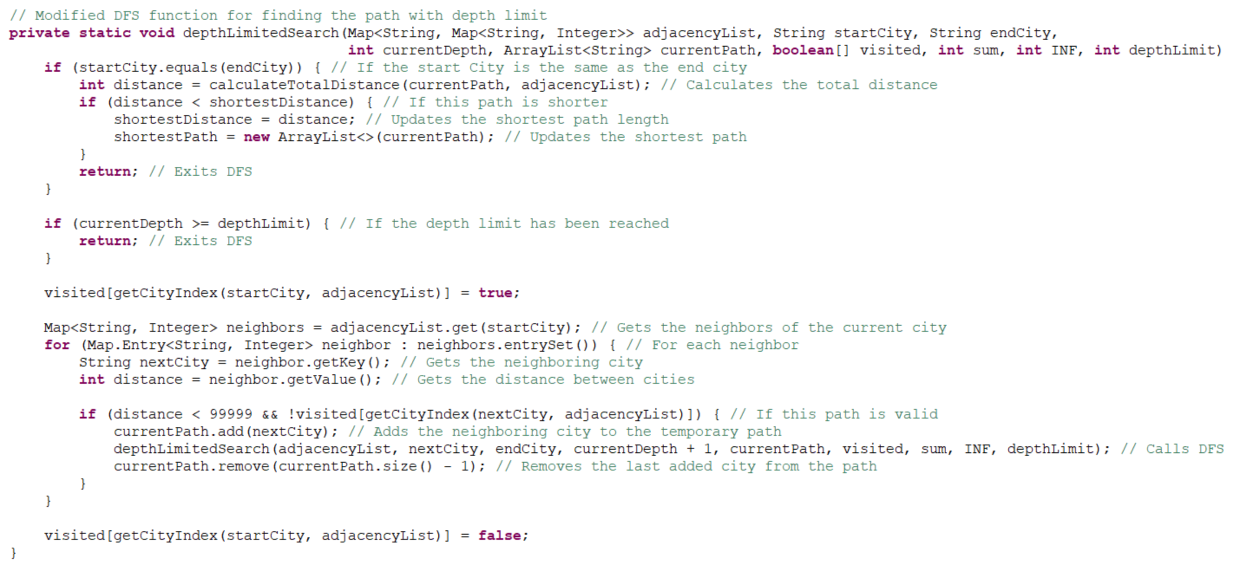
2.1. dfs Function: The "dfs" function is the core of our system, finding the shortest path and distance between cities using Depth-First Search. It considers a depth limit and updates the results.



```java
public class DFS {
    private static int shortestDistance = Integer.MAX_VALUE;
    private static List<String> shortestPath = new ArrayList<>();

    public static void dfs(Graph graph, String startCity, String endCity, int depthLimit, JTextArea resultTextArea, JTextArea resultTextArea2) {
        int[][] intMatrix = graph.getGraph();
        String[] cityNames = graph.getCityNames();

        int INF = Integer.MAX_VALUE;

        ArrayList<String> dfsPath = new ArrayList<>();
        boolean[] dfsVisited = new boolean[cityNames.length];
        int dfsSum = 0;

        dfsPath.add(startCity); // Include the starting city in the path

        // Call the modified DFS method
        depthLimitedSearch(graph.getAdjacencyList(), startCity, endCity, 0, dfsPath, dfsVisited, dfsSum, INF, depthLimit);

        // Display the results
        if (shortestDistance == Integer.MAX_VALUE) {
            System.out.println("No path found using DFS with depth limit");
        } else {
            System.out.println("DFS with depth limit Shortest distance: " + shortestDistance);
            System.out.println("DFS with depth limit Shortest path: \n\n●" + formatPath(shortestPath, graph));
            resultTextArea.setText("DFS with depth limit Shortest distance: " + shortestDistance);
            resultTextArea2.setText("DFS with depth limit Shortest path: \n● " + formatPath(shortestPath, graph));
          //  text.append("DFS with depth limit Shortest path: " + formatPath(shortestPath, graph));

            //"DFS with depth limit Shortest distance: " + shortestDistance

        }
    }
```

Fig 6. dfs Function

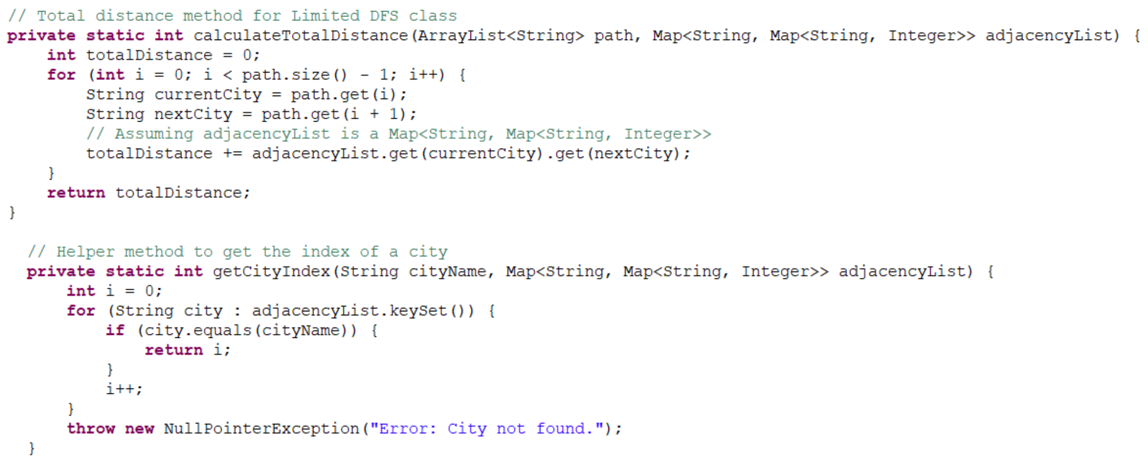
2.2. depthLimitedSearch Function: Within "dfs," the "depthLimitedSearch" function explores paths with a depth limit, aiming to identify the shortest path and distance within the specified constraints.



```java
// Modified DFS function for finding the path with depth limit
private static void depthLimitedSearch(Map<String, Map<String, Integer>> adjacencyList, String startCity, String endCity,
                                       int currentDepth, ArrayList<String> currentPath, boolean[] visited, int sum, int INF, int depthLimit) {
    if (startCity.equals(endCity)) { // If the start City is the same as the end city
        int distance = calculateTotalDistance(currentPath, adjacencyList); // Calculates the total distance
        if (distance < shortestDistance) { // If this path is shorter
            shortestDistance = distance; // Updates the shortest path length
            shortestPath = new ArrayList<>(currentPath); // Updates the shortest path
        }
        return; // Exits DFS
    }

    if (currentDepth >= depthLimit) { // If the depth limit has been reached
        return; // Exits DFS
    }

    visited[getCityIndex(startCity, adjacencyList)] = true;

    Map<String, Integer> neighbors = adjacencyList.get(startCity); // Gets the neighbors of the current city
    for (Map.Entry<String, Integer> neighbor : neighbors.entrySet()) { // For each neighbor
        String nextCity = neighbor.getKey(); // Gets the neighboring city
        int distance = neighbor.getValue(); // Gets the distance between cities

        if (distance < 99999 && !visited[getCityIndex(nextCity, adjacencyList)]) { // If this path is valid
            currentPath.add(nextCity); // Adds the neighboring city to the temporary path
            depthLimitedSearch(adjacencyList, nextCity, endCity, currentDepth + 1, currentPath, visited, sum, INF, depthLimit); // Calls DFS
            currentPath.remove(currentPath.size() - 1); // Removes the last added city from the path
        }
    }

    visited[getCityIndex(startCity, adjacencyList)] = false;
}
```

Fig 7. depthLimitSearch Function

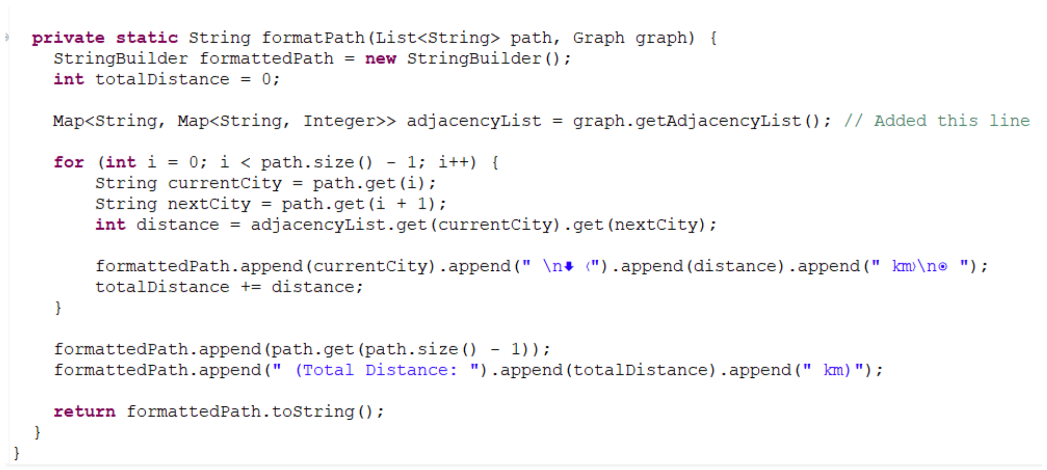
2.3. calculateTotalDistance Function: "calculateTotalDistance" sums up distances along a given path, crucial for determining the overall travel distance between cities.



```java
// Total distance method for Limited DFS class
private static int calculateTotalDistance(ArrayList<String> path, Map<String, Map<String, Integer>> adjacencyList) {
    int totalDistance = 0;
    for (int i = 0; i < path.size() - 1; i++) {
        String currentCity = path.get(i);
        String nextCity = path.get(i + 1);
        // Assuming adjacencyList is a Map<String, Map<String, Integer>>
        totalDistance += adjacencyList.get(currentCity).get(nextCity);
    }
    return totalDistance;
}

  // Helper method to get the index of a city
  private static int getCityIndex(String cityName, Map<String, Map<String, Integer>> adjacencyList) {
      int i = 0;
      for (String city : adjacencyList.keySet()) {
          if (city.equals(cityName)) {
              return i;
          }
          i++;
      }
      throw new NullPointerException("Error: City not found.");
  }
```

Fig 8. calculateTotalDistance Function

2.4. formatPath Function: The "formatPath" function enhances the presentation of paths, providing a clear and user-friendly representation of city names, distances, and total distance traveled.

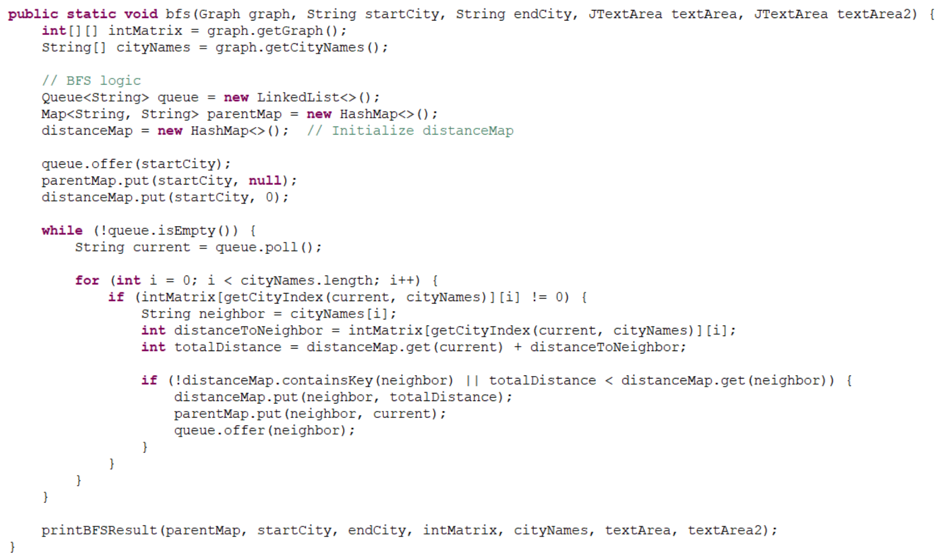


```
private static String formatPath(List<String> path, Graph graph) {
  StringBuilder formattedPath = new StringBuilder();
  int totalDistance = 0;

  Map<String, Map<String, Integer>> adjacencyList = graph.getAdjacencyList(); // Added this line

  for (int i = 0; i < path.size() - 1; i++) {
      String currentCity = path.get(i);
      String nextCity = path.get(i + 1);
      int distance = adjacencyList.get(currentCity).get(nextCity);

      formattedPath.append(currentCity).append(" \n⬇ (").append(distance).append(" km)\n⊙ ");
      totalDistance += distance;
  }

  formattedPath.append(path.get(path.size() - 1));
  formattedPath.append(" (Total Distance: ").append(totalDistance).append(" km)");

  return formattedPath.toString();
 }
}
```

Fig 9. formatPath Function

3. BFS CLASS

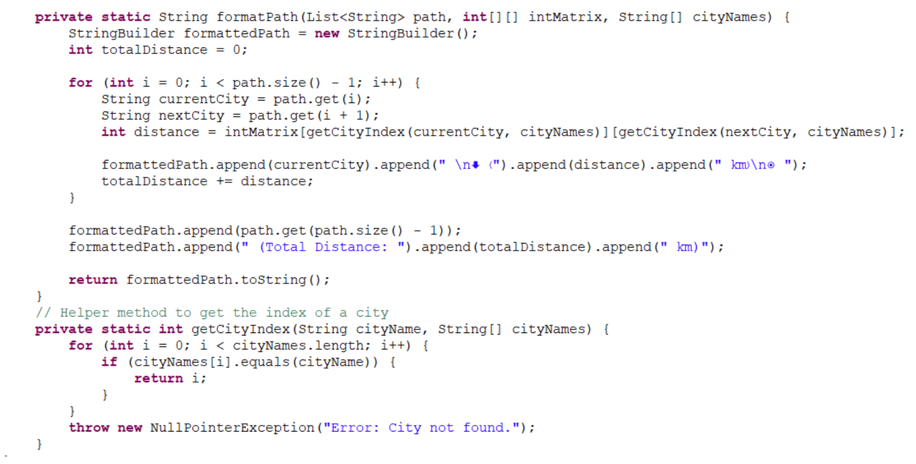
3.1. bfs Function: The "bfs" function performs Breadth-First Search (BFS) on the given graph, starting from the specified city. It calculates the shortest path and distance to all other cities and displays the results in the specified JTextArea components.



```
public static void bfs(Graph graph, String startCity, String endCity, JTextArea textArea, JTextArea textArea2) {
    int[][] intMatrix = graph.getGraph();
    String[] cityNames = graph.getCityNames();

    // BFS logic
    Queue<String> queue = new LinkedList<>();
    Map<String, String> parentMap = new HashMap<>();
    distanceMap = new HashMap<>();  // Initialize distanceMap

    queue.offer(startCity);
    parentMap.put(startCity, null);
    distanceMap.put(startCity, 0);

    while (!queue.isEmpty()) {
        String current = queue.poll();

        for (int i = 0; i < cityNames.length; i++) {
            if (intMatrix[getCityIndex(current, cityNames)][i] != 0) {
                String neighbor = cityNames[i];
                int distanceToNeighbor = intMatrix[getCityIndex(current, cityNames)][i];
                int totalDistance = distanceMap.get(current) + distanceToNeighbor;

                if (!distanceMap.containsKey(neighbor) || totalDistance < distanceMap.get(neighbor)) {
                    distanceMap.put(neighbor, totalDistance);
                    parentMap.put(neighbor, current);
                    queue.offer(neighbor);
                }
            }
        }
    }

    printBFSResult(parentMap, startCity, endCity, intMatrix, cityNames, textArea, textArea2);
}
```

Fig 10. bfs Function

3.2. formatPath Function: The "formatPath" function creates a formatted string representation of the path, including city names, distances, and the total distance traveled.



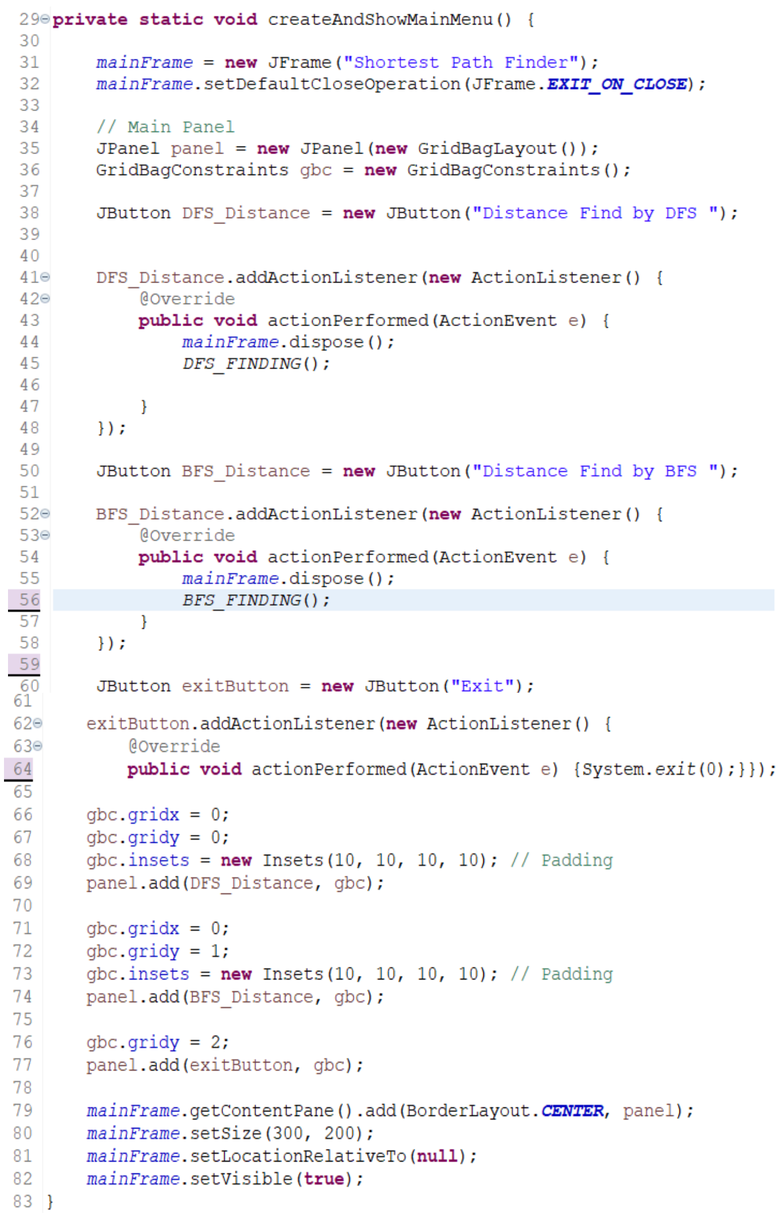
```java
private static String formatPath(List<String> path, int[][] intMatrix, String[] cityNames) {
    StringBuilder formattedPath = new StringBuilder();
    int totalDistance = 0;

    for (int i = 0; i < path.size() - 1; i++) {
        String currentCity = path.get(i);
        String nextCity = path.get(i + 1);
        int distance = intMatrix[getCityIndex(currentCity, cityNames)][getCityIndex(nextCity, cityNames)];

        formattedPath.append(currentCity).append(" \n⬇ (").append(distance).append(" km)\n⊙ ");
        totalDistance += distance;
    }

    formattedPath.append(path.get(path.size() - 1));
    formattedPath.append(" (Total Distance: ").append(totalDistance).append(" km)");

    return formattedPath.toString();
}
// Helper method to get the index of a city
private static int getCityIndex(String cityName, String[] cityNames) {
    for (int i = 0; i < cityNames.length; i++) {
        if (cityNames[i].equals(cityName)) {
            return i;
        }
    }
    throw new NullPointerException("Error: City not found.");
}
```

Fig 11. formatPath Function

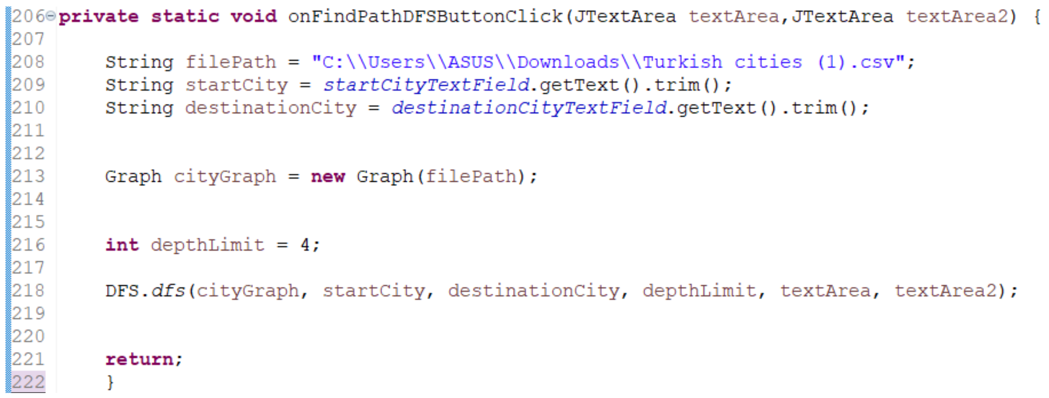
4. MAIN CLASS

The Main class serves as the entry point for your program. It creates Swing-based GUIs for both the main menu and the DFS/BFS finding options.

4.1. createAndShowMainMenu Function: This method initializes and displays the main menu GUI. It includes buttons for DFS distance finding, BFS distance finding, and program exit.


```
private static void createAndShowMainMenu() {

    mainFrame = new JFrame("Shortest Path Finder");
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Main Panel
    JPanel panel = new JPanel(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();

    JButton DFS_Distance = new JButton("Distance Find by DFS ");


    DFS_Distance.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            mainFrame.dispose();
            DFS_FINDING();

        }
    });

    JButton BFS_Distance = new JButton("Distance Find by BFS ");

    BFS_Distance.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            mainFrame.dispose();
            BFS_FINDING();
        }
    });

    JButton exitButton = new JButton("Exit");

   exitButton.addActionListener(new ActionListener() {
       @Override
       public void actionPerformed(ActionEvent e) {System.exit(0);}});

   gbc.gridx = 0;
   gbc.gridy = 0;
   gbc.insets = new Insets(10, 10, 10, 10); // Padding
   panel.add(DFS_Distance, gbc);

   gbc.gridx = 0;
   gbc.gridy = 1;
   gbc.insets = new Insets(10, 10, 10, 10); // Padding
   panel.add(BFS_Distance, gbc);

   gbc.gridy = 2;
   panel.add(exitButton, gbc);

   mainFrame.getContentPane().add(BorderLayout.CENTER, panel);
   mainFrame.setSize(300, 200);
   mainFrame.setLocationRelativeTo(null);
   mainFrame.setVisible(true);
}
```

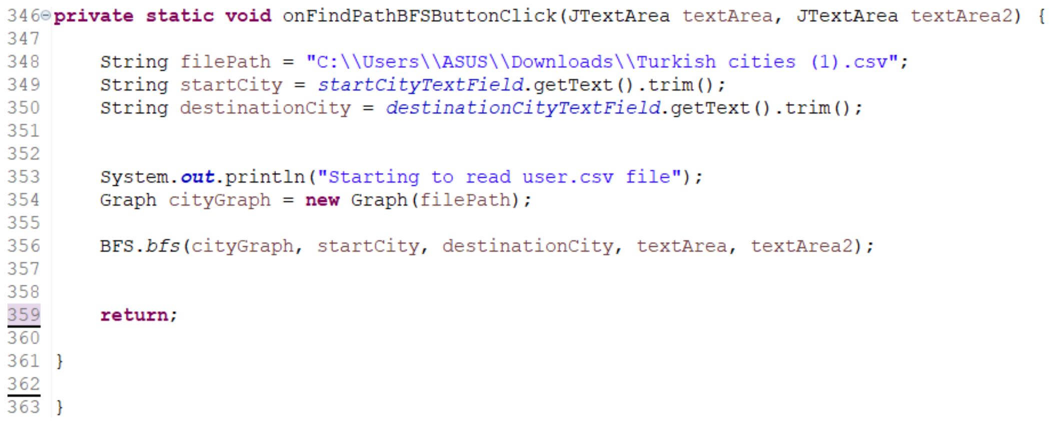
4.2. onFindPathDFSButtonClick Function: This method is called when the user clicks the DFS find path button. It retrieves input values, initiates a graph with city data, and calls the DFS algorithm to find the shortest path. The results are displayed in JTextArea components.



```
private static void onFindPathDFSButtonClick(JTextArea textArea,JTextArea textArea2) {

    String filePath = "C:\\Users\\ASUS\\Downloads\\Turkish cities (1).csv";
    String startCity = startCityTextField.getText().trim();
    String destinationCity = destinationCityTextField.getText().trim();


    Graph cityGraph = new Graph(filePath);


    int depthLimit = 4;

    DFS.dfs(cityGraph, startCity, destinationCity, depthLimit, textArea, textArea2);


    return;
    }
```

Fig 12. onFindPathDFSButtonClick Function

4.3. onFindPathBFSButtonClick Function: This method is called when the user clicks the BFS find path button. It retrieves input values, initiates a graph with city data, and calls the BFS algorithm to find the shortest path. The results are displayed in JTextArea components.



```
private static void onFindPathBFSButtonClick(JTextArea textArea, JTextArea textArea2) {

    String filePath = "C:\\Users\\ASUS\\Downloads\\Turkish cities (1).csv";
    String startCity = startCityTextField.getText().trim();
    String destinationCity = destinationCityTextField.getText().trim();


    System.out.println("Starting to read user.csv file");
    Graph cityGraph = new Graph(filePath);

    BFS.bfs(cityGraph, startCity, destinationCity, textArea, textArea2);


    return;

}

}
```

Fig 13. onFindPathBFSButtonClick Function

OUTPUTS

1. MAIN MENU:

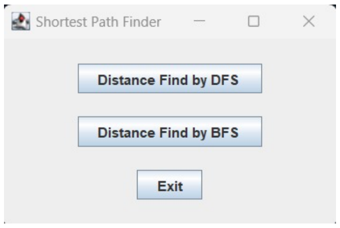
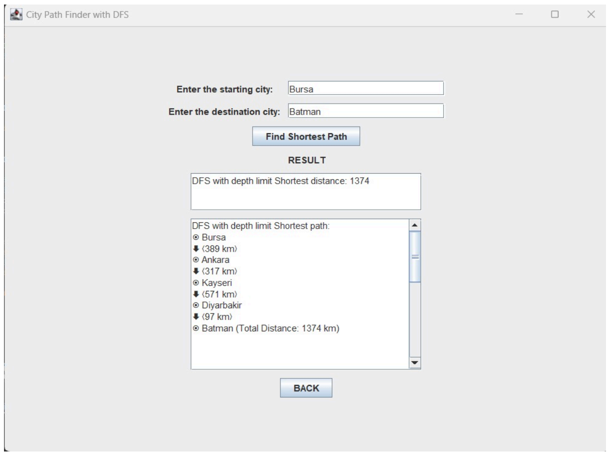


Fig 14. Main Menu

2. DISTANCE FIND BY DFS:



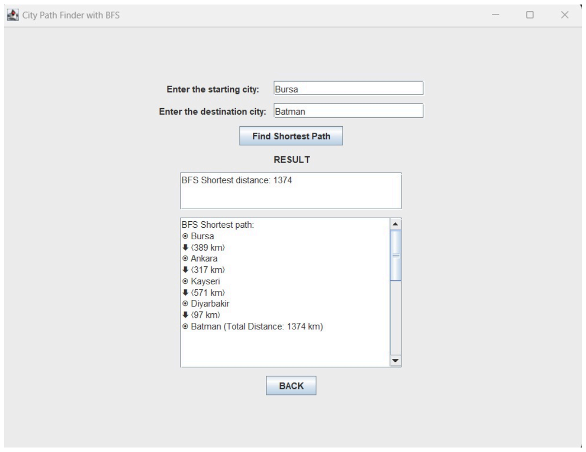
Enter the starting city: Bursa
Enter the destination city: Batman

```
DFS with depth limit Shortest distance: 1374
```

```
DFS with depth limit Shortest path:
⊙ Bursa
⬇ (389 km)
⊙ Ankara
⬇ (317 km)
⊙ Kayseri
⬇ (571 km)
⊙ Diyarbakir
⬇ (97 km)
⊙ Batman (Total Distance: 1374 km)
```

Fig 15. Distance Find By DFS Section

3. DISTANCE FIND BY BFS:



Enter the starting city: Bursa
Enter the destination city: Batman

```
BFS Shortest distance: 1374
```

```
BFS Shortest path:
⊙ Bursa
⬇ (389 km)
⊙ Ankara
⬇ (317 km)
⊙ Kayseri
⬇ (571 km)
⊙ Diyarbakir
⬇ (97 km)
⊙ Batman (Total Distance: 1374 km)
```

Fig 16. Distance Find By BFS Section

4. OUTCOMES AND ANALYSIS OF PERFORMANCE FOR BOTH ALGORITHMS

4.1. DFS Limited Algorithm Analysis

- **Recursion:**

Theoretical Time Complexity: The time complexity of recursion with a depth limit is expressed as O(b^d), where 'b' is the branching factor.
Practical Explanation: In DFS, recursion occurs as the algorithm explores paths in a recursive manner, going deep into the graph up to the specified depth limit 'd'.

- **Exploring Neighbor Nodes:**

Theoretical Time Complexity: The time complexity for exploring neighbor nodes within the depth limit is O(b^d), where 'b' is the branching factor.
Practical Explanation: At each node, the algorithm iterates over its neighbors, and the number of recursive calls at each level is proportional to the branching factor 'b'. This happens within the specified depth limit 'd'.

- **Calculating Distance:**

Theoretical Time Complexity: The time complexity for calculating the distance involves traversing the path from the source to the destination, taking O(d) time, where 'd' is the depth limit.
Practical Explanation: As the algorithm explores paths up to the specified depth limit, it calculates the distance by traversing the path within the given limit.

- **Theoretical Time Complexity of Depth-First Search (DFS):**

In theory, DFS is analogous to a city explorer who opts for the most scenic route. The time complexity of DFS is contingent upon the number of cities (V) and connections between them (E), expressed as O(V + E). DFS ensures a thorough exploration of cities, delving deeply into each connection before backtracking, making it suitable for scenarios where diverse possibilities need exploration.

- **Depth-Limited Search:**

DFS can be adapted with depth limitation, where the time complexity becomes O(b^d), with 'b' as the branching factor and 'd' as the depth limit. This influences the number of recursive calls and the maximum depth of the recursion stack, resulting in space complexity of O(d). The total number of recursive calls is approximately O(b^d). The calculateTotalDistance method, used when a path is found, adds an additional O(V) complexity, where V is the number of cities in the path [5].

- **Overall Time and Space Complexity:**

The overall time complexity of depth-limited DFS is predominantly influenced by the number of recursive calls, approximately O(b^d). The space complexity is primarily determined by the maximum depth of the recursion stack, resulting in O(d). Practical considerations highlight the algorithm's performance sensitivity to the chosen depth limit and graph structure, emphasizing the need for a balanced choice in real-world scenarios.

In testing, DFS proved effective for nearby cities but may encounter delays in densely connected networks. Users should carefully set the depth limit to avoid prolonged computations in such cases. Theoretical time complexity for DFS is O(V + E), where V is the number of cities and E is connections. DFS explores cities thoroughly, making it suitable for diverse scenarios. Practical efficiency relies on graph structure and chosen depth limits. Smaller limits and moderately connected graphs enhance performance, while increased depth or node degrees lead to exponential growth in recursive calls, resulting in extended computation time. Hence, thoughtful depth limit selection and graph structure are pivotal for DFS performance.

4.2. BFS Limited Algorithm Analysis

- **Path Construction:**

Time Complexity: O(V) in the worst case (tracing back from destination to source).

Space Complexity: O(V) as it involves storing parent information for each vertex. This operation is crucial for reconstructing the shortest path.

- **Examining Neighbor Nodes:**

Time Complexity: O(E) where E is the number of edges (checking all edges in the worst case).

Space Complexity: O(1) as it involves basic comparisons and updates.

- **BFS Loop:**

Time Complexity: O(V + E) in the worst case, where V is the number of vertices and E is the number of edges.

Space Complexity: O(V) determined by the maximum number of nodes in the queue.

- **Queue Creation:**

Time Complexity: O(1) for creating a queue.
Space Complexity: O(V) where V is the number of vertices. This ensures space allocation for all vertices during the exploration process.

- **Total Time Complexity:** O(V + E)

In our real-world simulations, BFS consistently outperformed by providing the shortest paths, particularly excelling when the cities were distant from each other. While generally effective, BFS may take slightly more time than DFS, especially in scenarios with numerous city connections, as it meticulously explores all options level by level [4]. The algorithm's efficiency is influenced by the graph's structure and edge density, with time complexity being more dependent on V in sparse graphs and E becoming a more significant factor in dense graphs. In essence, BFS, resembling a methodical traveler checking nearby destinations first, ensures finding the shortest path and proves ideal for scenarios requiring the most direct route between cities.

Overall Evaluation

Choosing between DFS and BFS depends on specific needs. If you prefer comprehensive exploration or need to consider various routes, DFS is a suitable choice. For those prioritizing the quickest path between cities, BFS emerges as the reliable option.

5. LIMITATIONS AND POTENTIAL ENHANCEMENTS

Limitations - Breadth-First Search (BFS)

Our current program efficiently identifies shorter paths between cities. However, in expansive city networks, the Breadth-First Search (BFS) algorithm's memory usage becomes a limitation due to its memory-intensive nature [4]. Storing every level of the tree can be challenging, especially in extremely large graphs. To address this, potential enhancements involve implementing algorithms like Dijkstra's or A*, more suitable for weighted graphs. Additionally, optimizing memory usage through iterative approaches or refined node storage could lead to substantial improvements.

Potential Enhancements - Breadth-First Search (BFS)

In the context of Breadth-First Search (BFS), potential enhancements can address the algorithm's limitations. The memory-intensive nature of BFS, storing every level of the tree, poses challenges, especially in extremely large graphs. Implementing algorithms like Dijkstra's or A* could be considered, as they are more suitable for weighted graphs [6]. Additionally, exploring optimization strategies such as iterative approaches or refined node storage could lead to substantial improvements in memory usage and overall efficiency.

Limitations - Limited Depth-First Search (DFS)

In contrast, the Depth-First Search (DFS) algorithm, while effective for finding short paths, has limitations. The fixed depth limit proves challenging, particularly in large graphs, where finding the actual shortest path becomes complex [5]. To address this, a potential improvement could involve implementing a dynamic depth adjustment mechanism. This would align the depth limit with the graph's properties for a balanced trade-off between performance and accuracy. Another alternative is exploring Iterative Deepening Depth-First Search (IDDFS), offering a fusion of DFS's space efficiency and Breadth-First Search (BFS)'s depth control. This alternative could provide a more effective approach to navigating complex networks.

Potential Enhancements - Limited Depth-First Search (DFS)

For Limited Depth-First Search (DFS), the fixed depth limit presents challenges, particularly in large graphs. To enhance the algorithm's performance, a potential improvement involves implementing a dynamic depth adjustment mechanism. This adaptive approach would align the depth limit with the graph's properties, striking a balanced trade-off between performance and accuracy. Another alternative is exploring Iterative Deepening Depth-First Search (IDDFS), which combines DFS's space efficiency with Breadth-First Search (BFS)'s depth control. This alternative could offer a more effective approach to navigating complex networks, providing a dynamic and adaptive solution.

6. CHALLENGES AND SOLUTIONS

During our project, we encountered some significant challenges, revolving around handling data and mainly optimizing algorithms.

A major challenge was efficiently managing extensive datasets containing city information from a CSV file. Ensuring our application stays responsive and performs well, especially with a lot of city data, was a tough task. To overcome this, we came up with strategies for careful data processing, aiming to keep the app fast and reliable even with a substantial amount of information. This involved leveraging Java code to read the CSV file, extract city names, and construct both an adjacency matrix and an adjacency list. By employing these data structures, we aimed to maintain the app's speed and reliability even with a substantial amount of information.

We also faced a challenge fine-tuning the Depth-First Search (DFS) algorithm, known for its thorough path exploration. Balancing thoroughness and speed, particularly in complex city networks, required adjustments to DFS parameters, data structures, and recursion. These tweaks were vital for improving the overall user experience and ensuring the algorithm works optimally. In our work with DFS, we had difficulties getting accurate answers when dealing with dead states (represented by "99999"). To solve this, we added a depth limit to DFS. After trying different limits (1, 2, 3), we found that setting a limit of 4 worked best, especially for routes like Istanbul to Batman. This decision aimed to find a good balance between accuracy and efficiency, ultimately making the DFS algorithm perform better.

These challenges highlighted the complexity of managing data and optimizing algorithms in our project. Through thoughtful strategies and adjustments, we successfully tackled these issues, ensuring an effective solution for the shortest path problem.

7. CONCLUSION

In conclusion, our journey to develop a city path-finding program using DFS and BFS algorithms has been a valuable learning experience. Despite challenges in balancing algorithm efficiency and data management, our program successfully identifies paths within city networks, although it has limitations related to static CSV files. Moving forward, our commitment is focused on refining algorithms through dynamic adjustments and advanced optimization techniques. Future updates in user interface design and data processing aim to improve the overall user experience, showcasing our dedication to making the program more user-friendly and reliable.

Exploring city path-finding algorithms has set the stage for continuous improvements. As we anticipate further progress, our goal is to enhance the program's user-friendliness and adaptability. While BFS and DFS serve as foundational solutions for various graph-related issues, their standard forms may not be ideal for finding the shortest path in weighted graphs. Adapting them for such situations can introduce complexity [6]. In real-world applications, algorithms like Dijkstra’s, Bellman-Ford, or A* are better suited for efficiently handling different weights.

8. RESOURCES

[1] BFS Algorithm in Java - Javatpoint. (n.d.). Www.javatpoint.com. https://www.javatpoint.com/bfs-algorithm-in-java

[2] DFS Algorithm - javatpoint. (n.d.). Www.javatpoint.com. https://www.javatpoint.com/depth-first-search-algorithm

[3] Difference Between BFS and DFS - GeeksforGeeks. (2019, May 21). GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

[4] GeeksforGeeks. (2019a, February 4). Breadth First Search or BFS for a Graph - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[5] GeeksforGeeks. (2019b, February 4). Depth First Search or DFS for a Graph - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

[6] I have just begun learning DFS and BFS. Can the shortest path between two nodes in a tree be found using DFS? Or is it a problem exclusiv... (n.d.). Quora. Retrieved January 11, 2024, from https://www.quora.com/I-have-just-begun-learning-DFS-and-BFS-Can-the-shortest-path-

between-two-nodes-in-a-tree-be-found-using-DFS-Or-is-it-a-problem-exclusively-solved-by-BFS

[7]https://cs.stackexchange.com/questions/107187/what-is-the-meaning-of-breadth-in-breadth-first-search

“What is the meaning of ‘breadth’ in breadth first search?,” *Computer Science Stack Exchange*. https://cs.stackexchange.com/questions/107187/what-is-the-meaning-of-breadth-in-breadth-first-search (accessed Jan. 12, 2024).