
# COMP201 - Data Structures and Algorithms

## Term Project: Shortest Path Problem Report


| | |
|---|---|
| **Tolga Neslioğlu** | **042301100** |
| **Cem Alp Özer** | **042301097** |
| **Efe Ersöz** | **042202008** |
| **Yağmur Kaykaç** | **041801112** |
| **Beyza Zerdalı** | **042201050** |

**MEF University, Istanbul, Turkey**


**Date: 19.12.2024**

# Introduction

This project proposes possible solutions to the shortest path finding problem using DFS (Depth First Search) and BFS (Breadth First Search) algorithms. Implementations and the comparison of the two algorithms are evaluated considering both theoretical and empirical time complexities. Please note that the BFS and DFS algorithms implemented in this project is different from their standard versions. In a typical BFS, the algorithm explores level by level and guarantees the shortest path in an unweighted graph but not in a weighted graph. On the other hand, a standard DFS explores deeper paths first and does not guarantee the shortest path. In this project, we tried to modify these algorithms to handle weighted graphs and to make sure they identify the shortest path between a source and a destination.

All the data structures needed to implement the BFS and DFS algorithms, such as Linked List, Stack, and Queue, were created from scratch without using basic Java libraries. We attempted to guarantee that these implementations were both time and memory efficient. Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser's book Data Structures and Algorithms in Java, 6th Edition, Addison-Wesley, provided assistance.

The challenges that came up during the implementation of the BFS and DFS algorithms and the solutions that are implemented to solve these problems, such as implementation of a method to import given city distances adjacency matrix into a two-dimensional array that Java can read are also explained while highlighting all the reasons of the chosen solution method.

This report begins with the definition of the problem. Then proceeds with the description of the solutions using the DFS and BFS algorithms adapted to the given problem, showing their output. Outputs include empirical time complexity of each algorithm. Finally, an empirical time comparison of both DFS and BFS algorithms is performed.

### Definition of the Problem:

The shortest Path problem is based on the cities located in Türkiye. We have some of the Turkish cities, some of them connected directly whereas some of them connected indirectly to each other. To demonstrate the network structure, we use a table which is in the form of a matrix as seen in Figure 1.

| | Istanbul | Ankara | Izmir | Bursa | Adana | Gaziantep | Konya | Diyarbakir | Antalya | Mersin | Kayseri | Urfa | Malatya | Samsun | Denizli | Batman | Trabzon |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Istanbul** | 0 | 449 | 99999 | 153 | 99999 | 99999 | 645 | 99999 | 690 | 956 | 776 | 99999 | 99999 | 737 | 649 | 99999 | 99999 |
| **Ankara** | | 0 | 591 | 389 | 484 | 705 | 266 | 1003 | 483 | 501 | 317 | 848 | 682 | 402 | 483 | 99999 | 732 |
| **Izmir** | | | 0 | 333 | 898 | 1118 | 560 | 99999 | 451 | 911 | 874 | 99999 | 99999 | 1003 | 238 | 99999 | 99999 |
| **Bursa** | | | | 0 | 856 | 1075 | 507 | 99999 | 546 | 869 | 715 | 1212 | 1049 | 750 | 480 | 99999 | 1091 |
| **Adana** | | | | | 0 | 225 | 346 | 525 | 649 | 95 | 307 | 369 | 389 | 719 | 758 | 618 | 851 |
| **Gaziantep** | | | | | | 0 | 568 | 315 | 785 | 311 | 357 | 151 | 251 | 803 | 974 | 409 | 838 |
| **Konya** | | | | | | | 0 | 866 | 303 | 360 | 306 | 702 | 729 | 663 | 386 | 964 | 896 |
| **Diyarbakir** | | | | | | | | 0 | 99999 | 610 | 571 | 182 | 235 | 803 | 1276 | 97 | 586 |
| **Antalya** | | | | | | | | | 0 | 631 | 610 | 924 | 99999 | 99999 | 217 | 99999 | 99999 |
| **Mersin** | | | | | | | | | | 0 | 99999 | 99999 | 99999 | 99999 | 99999 | 99999 | 99999 |
| **Kayseri** | | | | | | | | | | | 0 | 99999 | 99999 | 99999 | 99999 | 99999 | 99999 |
| **Urfa** | | | | | | | | | | | | 0 | 99999 | 99999 | 99999 | 99999 | 99999 |
| **Malatya** | | | | | | | | | | | | | 0 | 99999 | 99999 | 99999 | 99999 |
| **Samsun** | | | | | | | | | | | | | | 0 | 99999 | 858 | 99999 |
| **Denizli** | | | | | | | | | | | | | | | 0 | 99999 | 99999 |
| **Batman** | | | | | | | | | | | | | | | | 0 | 99999 |
| **Trabzon** | | | | | | | | | | | | | | | | | 0 |

*Figure 1. Matrix of the Cities of Türkiye*

### Identifying Pathways:

To compute each path, we need to obtain a starting point, and the destination point from the user. Each point is set from the cities in the matrix in Figure 1. The program begins from computing from the starting point, which is one of the cities in the given matrix, towards the destination point by testing every single element of the matrix until it reaches the destination point or the end points shown in red color (99999s).

## Algorithm and Data Structure Descriptions

The algorithms used to solve this problem are Depth-First Search (DFS) and Breadth-First Search (BFS). Both are used to traverse graphs, although their implementations and behavior are different.

DFS uses a stack to keep track of the paths it explores, either explicitly or via recursion. This means it always processes the most recently added path first, following the Last In, First Out (LIFO) principle. BFS, on the other hand, uses a queue to manage its paths. It processes paths in the order they were added, moving level by level through all connected nodes using the First In, First Out (FIFO) principle. A basic BFS algorithm can identify the shortest path in an unweighted graph by exploring all neighbors at the current depth before proceeding deeper. In comparison, DFS does not guarantee the shortest path because it investigates deeper paths before others. However, standard BFS and DFS cannot handle weighted graphs without modifications to account for edge weights.

## Java Program Structure

The overall architecture of the Java program is designed to efficiently solve the shortest path problem using both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms. The program is modular, with each component responsible for a specific part of the functionality. Here is a detailed description of the main classes and methods:

### 1. `CreateArray Class:`

- **Purpose:** Reads the city names and distances from the given CSV file and stores them in appropriately sized arrays.
- **Methods:**
  - `getHeaderFromCsv(File filePath):` Reads the header (city names) from the CSV file.
  - `getDistancesFromCsv(File filePath):` Reads the distances between cities and stores them in a 2D integer array.

### 2. `Node Class:`

- **Purpose:** Represents a node in the linked list used for stack and queue implementations.
- **Attributes:**
  - `element`: The data stored in the node.
  - `next`: Reference to the next node in the list.

### 3. `SinglyLinkedList Class:`

- **Purpose:** Implements a singly linked list to be used by stack and queue classes.
- **Methods:**
  - `addFirst(E e):` Adds an element to the front of the list.
  - `addLast(E e):` Adds an element to the end of the list.
  - `removeFirst():` Removes and returns the first element.
  - `removeLast():` Removes and returns the last element.

**4. `LinkedStack Class:`**

- **Purpose:** Implements a stack data structure using a linked list.
- **Methods:**
  - `push(E data):` Adds an element to the top of the stack.
  - `pop():` Removes and returns the top element of the stack.
  - `top():` Returns the top element without removing it.
  - `isEmpty():` Checks if the stack is empty.
  - `size():` Returns the number of elements in the stack.

**5. `LinkedQueue Class:`**

- **Purpose:** Implements a queue data structure using a linked list.
- **Methods:**
  - `enqueue(E data):` Adds an element to the end of the queue.
  - `dequeue():` Removes and returns the front element of the queue.
  - first(): Returns the front element without removing it.
  - `isEmpty():` Checks if the queue is empty.
  - `size():` Returns the number of elements in the queue.

**6. `DFS Class:`**

- **Purpose:** Implements the DFS algorithm to find the shortest path in a weighted graph.
- **Methods:**
  - `shortestPathDFS (int start, int arrival, String[] citiesHeader, int[][] distancesArray)`: Recursively explores all paths from the start city to the arrival city using DFS.
  - `isForbidden(String city, LinkedStack<String> stack):`
    The `isForbidden` method checks if a city has already been visited during the DFS traversal.

**7. `BFS Class:`**

- **Purpose:** Implements the BFS algorithm to find the shortest path in a weighted graph.
- **Methods:**
  - `shortestPathBFS(int start, int destination):` Finds the shortest path from the source to the destination using BFS.
  - `pathLength(ArrayList<Integer> path):` A method to calculate the lenght of a path.

# Implementation Details

**DFS Class Overview:**

The DFS class implements the Depth First Search (DFS) algorithm with backtracking to find the shortest path between two nodes in a graph. This approach recursively explores all possible paths and prunes branches that exceed the current shortest path length.

**Data Structures:**

- `static String shortestPath:` Stores the shortest path as a String
- `static int shortestDistance:` Stores the shortest path distance.
- `static LinkedStack<String> citiesStack:` Maintains the current path as a stack.
- `static SinglyLinkedList<Integer> distances:` Linked list to store distances of the current path.
- `static LinkedStack<String> visited:` Stack to track visited nodes.

➢ `shortestPathDFS():`

Computes the shortest path between a start and an arrival node using DFS.

- **Input Parameters:**
  - `(int) start` and `(int) arrival:` Indices of the start and destination nodes.
  - `(String[]) citiesHeader`: Array of city names.
  - `(int[][]) distancesArray`: 2D array representing distances between nodes.
- **Logic**:
  - The current city is added to a stack to represent the path being explored.
  - Total distance for the current path is calculated by summing values in the distances linked list.
  - Pruning: If the current path length exceeds the shortest known distance, the method backtracks.
  - If the destination node is reached, the current path is evaluated for shortest distance and stored if it's optimal.
  - Recursive calls explore all unvisited neighbors of the current node, using the `isForbidden()` method to check if a node has already been visited.
  - Backtracking ensures all explored nodes and distances are removed from their respective stacks/lists after processing.

- `isForbidden():`

  Checks if a city is already in the visited stack.

  - **Input Parameters:**
    - `LinkedStack<String> visited:` Stack to track visited nodes.
    - `String city:` Checks if this city is in the stack or not.
  - **Logic:**
    - Iterates through the linked list representation of the stack to check for the presence of the city.

**BFS Class Overview:**

The BFS class implements the Breadth First Search (BFS) algorithm to find the shortest path between two nodes in a given adjacency matrix. Our approach uses a queue to explore paths level by level, ensuring the shortest path is found.

**Data Structures:**

- `LinkedQueue<ArrayList<Integer>> open:` Stores paths during exploration.
- `int[][] adjMatrix:` Represents the graph as an adjacency matrix.
- `ArrayList<Integer> startingPath:` Stores the shortest path found.
- `String path:` Contains a formatted representation of the shortest path.

**Constructor:**

- `BFS(int[][] adjMatrix, String[] vertices):`

  Initializes the adjacency matrix (`adjMatrix`) and vertex labels (`vertices`) for the graph.

**Methods:**

- `shortestPathBFS():`

  Computes the shortest path between a start and a destination node using BFS.

  - **Input Parameters:**
    - `(int) start` and `(int) destination`: Indices of the source and destination nodes.

- **Logic**:
    - A queue (**`open`**) stores paths represented as lists of node indices.
    - A starting path containing the source node is enqueued.
    - While the queue is not empty, the method dequeues the current path and evaluates it:
        - If the path's total distance exceeds the shortest known path, it is skipped.
        - If the destination is reached, the shortest path and its length are updated.
        - Otherwise, unvisited neighbors of the current node are appended to the path, and the new path is enqueued.
    - The method terminates once all possible paths are explored, leaving the shortest path in **`shortestPath`**.

➢ **`pathlength():`**

Calculates the total distance of a given path. This method is used in our BFS algorithm to optimize by comparing the current path's distance with the shortest known path's distance. We used this method for calculating the distances.

- **Logic:**
    - Iterates through pairs of consecutive nodes in the path, summing the weights of the edges between them.

# Complexity Analysis:

## Theoretical Time Complexity

### DFS with Recursion

The DFS algorithm is designed to find the shortest path using a recursive depth-first approach combined with backtracking. Its theoretical time complexity can be broken down as follows:

- **Path Exploration:**
  In the worst-case scenario, the DFS algorithm may explore all possible paths in the graph. If the graph contains n nodes and the edges are densely connected, this may result in O(n!) potential paths. This factorial growth arises because the algorithm may attempt permutations of nodes before pruning unviable paths.
- **Operations Per Path:**
  - **Checking Visited Nodes:** Each recursive call involves checking the visited nodes, which takes O(n) due to the use of a linked stack.
  - **Summing Distances:** Calculating the total distance of the current path involves iterating over the distances linked list, which also takes O(n)
  - **Neighbor Exploration:** For each node, the algorithm explores all its neighbors. This involves adjacency matrix lookups, which take O(n).

Combining these, the worst-case theoretical complexity of the DFS is approximated as O(n!)

- **Pruning Optimization:**
  The pruning mechanism significantly reduces the search space, also resulting significantly faster completion time. By discarding paths that exceed the current shortest distance, the algorithm avoids exploring many unnecessary permutations, leading to better runtime performance in realistic scenarios.

**BFS with Path Optimization**

The BFS algorithm employs a brute-force approach, exploring all possible paths from the source to the destination, with some optimizations for pruning longer paths early.

- **Path Exploration:**
  Similar to DFS, BFS explores paths in the graph. However, BFS uses a queue to handle paths iteratively rather than recursively. In dense graphs, the number of paths can again grow factorially, resulting in O(n!) potential paths.
- **Operations Per Path:**
  - **Dequeuing a Path:** Removing a path from the queue is O(1).
  - **Checking Neighbors:** For each node, BFS checks all its neighbors, which involves O(n) operations due to the adjacency matrix.
  - **Path Updates:** Each new path is created by copying the current path and appending a neighbor, which takes O(n).

Thus, the worst-case complexity of BFS is also O(n!).

- **Pruning Optimization:**
  The optimization mechanism in BFS, which skips paths exceeding the current shortest path length, greatly improves practical runtime. BFS's iterative nature often makes it more memory-intensive but slightly easier to manage computationally compared to DFS.

---

**Empirical Observations from Runtime Analysis**

In testing with the given adjacency matrix, both algorithms showed significant performance improvements due to their pruning mechanisms. Key observations include:

1. **DFS Performance:**
   - Even without the pruning optimization, DFS algorithm demanded less system source, since only one stack is stored on the memory, which does not cause heap overflow (out of memory) exceptions.

2. **BFS Performance:**
    - BFS outperformed DFS simple and shorter paths, such as the cities that have direct connection in-between.
    - Without the pruning optimization, BFS algorithm becomes impossible to run until it finds all the paths (with the given size matrix) because very big number of queues that increases more and more are stored in the heap at same time, which causes heap overflow eventually.

## Results and Analysis

Based on the implementations of Breadth-First Search (BFS) and Depth-First Search (DFS) and the provided chart comparing their elapsed times for various routes, we can draw several conclusions about their performance and suitability for different scenarios.

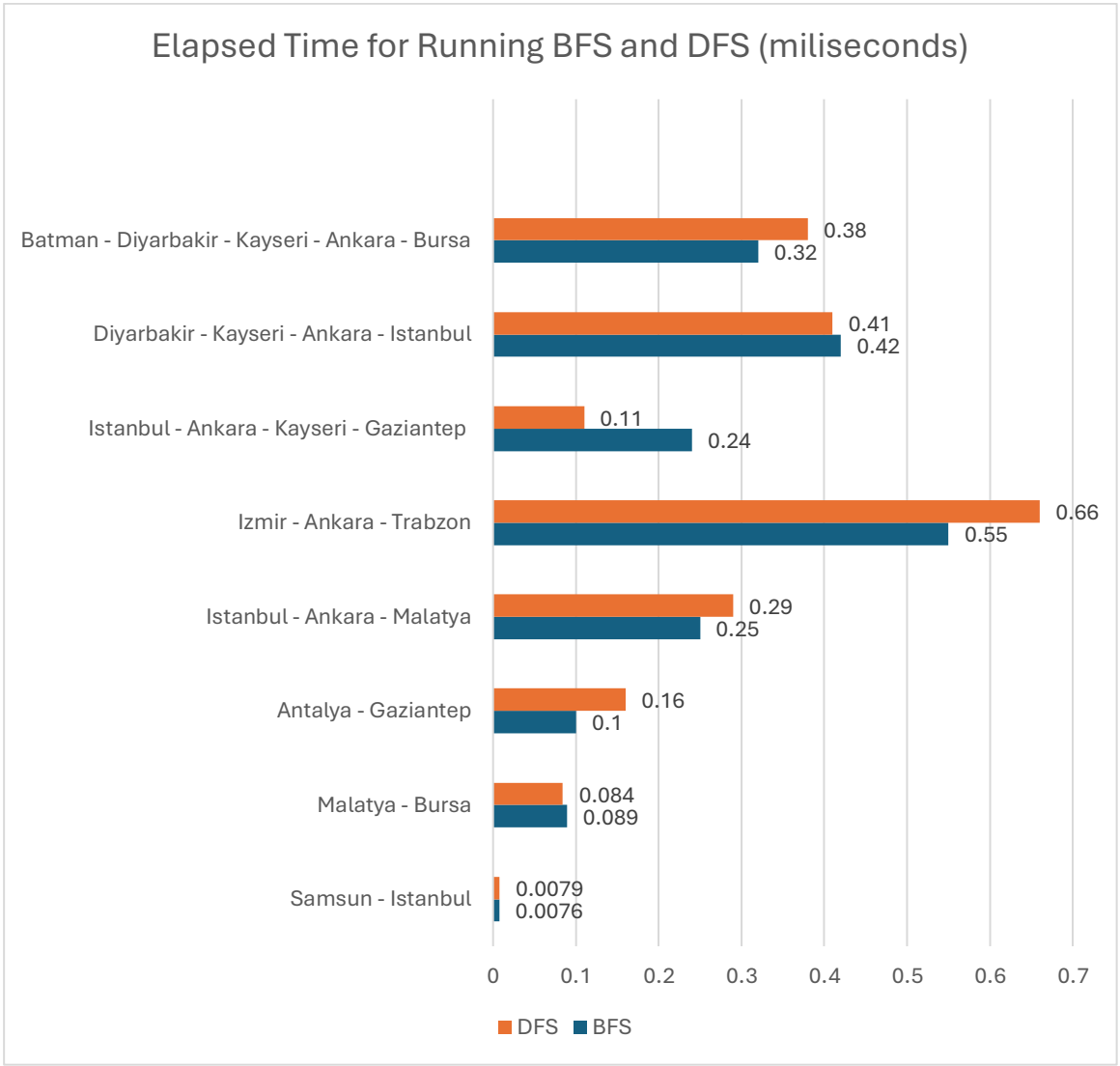
BFS, with its level-wise exploration strategy, ensures that the shortest path is always found. Its theoretical time complexity is O(n!) in the worst-case scenario, primarily due to the potential factorial number of paths in dense graphs. However, in practice, the algorithm performs much faster due to pruning and the optimization provided by the adjacency matrix structure. BFS demonstrates consistent performance across different routes, as observed in the chart. For example, even for complex routes like " Batman - Diyarbakir - Kayseri - Ankara - Bursa," BFS remains efficient despite slightly longer execution times compared to simpler routes.

DFS, on the other hand, is a recursive algorithm that explores deeper paths before backtracking. Its worst-case time complexity is also O(n!), as it can explore all possible paths before pruning. DFS can perform well in sparse graphs or scenarios where the target is located deeper within a specific path. The chart highlights that DFS exhibits significant variations in elapsed times. For instance, it performs well for routes like "Istanbul - Ankara - Malatya" likely due to effective pruning, but struggles with some routes, such as "Izmir - Ankara - Trabzon" where exploration of deeper paths leads to longer execution times.

From a performance perspective, BFS generally outperforms DFS in situations where finding the shortest path in short time is critical. This is evident in routes requiring a more exhaustive search through complex graph structures. Conversely, DFS is more suitable for tasks that prioritize deeper exploration or when memory usage needs to be minimized, as it operates on a single path at a time.

Each algorithm has its advantages and disadvantages. Modified BFS is ideal for problems that require shortest paths in short amount of time, particularly in weighted or dense graphs, but it may consume more memory due to storing all paths at each level. DFS, on the other hand, is advantageous in scenarios with sparse graphs or where a quick exploration of deeper paths is beneficial. However, it risks redundant exploration and may fail to find the shortest path without proper optimization.

In conclusion, BFS and DFS have distinct strengths and trade-offs, making them suitable for different types of problems. BFS is recommended for scenarios requiring guaranteed shortest paths and optimal performance in complex graphs, while DFS is better suited for exploratory tasks or memory-constrained environments. These findings align with both the theoretical understanding of the algorithms and the empirical data from the chart.



## Discussion of Limitations and Improvements:

For both BFS and DFS algorithms, we were not able to run the algorithms until they finished exploring all the paths without the pruning optimization.

We were able to run DFS algorithm without having a heap overflow or out of memory exception since it is not very demanding on system sources. However, for the given sized dataset (city distances adjacency matrix) even after 20 minutes of runtime, the algorithm couldn't even manage to explore half of the available paths. It is evident that without very advanced hardware, it is not possible to complete exploring all the paths in a feasible amount of time for given dataset.

On the other hand, we weren't able to run BFS algorithm for that long without having heap overflow even with +10GB of heap space given. The algorithm is highly demanding on system resources, and it is almost impossible to run on an average PC without any optimizations.

For research purposes, the performance of both algorithms without pruning optimization could be measured on a smaller dataset. We did run both algorithms on smaller datasets to prove that algorithms were working, however, to be able to compare the performance of the algorithms without any optimizations, further experiments and tests need to be conducted.

The pruning optimization was a game changer for both algorithms, which reduced the runtime from hours to milliseconds. Pruning optimizations basically allows to stop exploring current path, if it is already longer than the shortest path found so far, to avoid unnecessary system source and time loss.

After pruning optimizations, BFS overperformed the DFS in runtime, finding the shortest path in shorter time in most cases, though it is not that significant. However, we were not able to use differently sized datasets, which could uncover different patterns. For the dataset we had, were not able to find a pattern to analyze when and why BFS outperforms the DFS algorithm. To uncover a pattern, datasets of different sizes should be used and the number of times the algorithms have run should be increased.

Since the BFS and DFS algorithm were written by different students, the implementation approaches were not identical, which may affect the performance of algorithms. The DFS uses LinkedList structure to store some data, while BFS uses `ArrayList` which is not identical in terms of performance. Also, the way that `toString()` methods were implemented causes DFS algorithm to run `toString()` method multiple times during the runtime, whereas BFS only runs it once, which definitely hinders the DFS's performance. To eliminate the uncontrolled

variables, further optimizations should be done on each method to eliminate inconsistencies during runtime and make a fair comparison.

It should also be mentioned that the algorithms were run on personal computers, which have many different tasks that are running in the background by the operating system. It caused us to get different and inconsistent results when giving the same inputs to the program. To minimize this, a more isolated OS environment could be used to get more consistent results in each run.

## Conclusion:

In this project, the shortest path problem for weighted graphs was addressed using modified versions of Depth-First Search (DFS) and Breadth-First Search (BFS). The implementations were evaluated both theoretically and empirically, with observations made on their performance under various conditions.

The DFS algorithm, adapted with backtracking, proved to be memory-efficient due to its use of a single stack. Its ability to prune paths that exceed the current shortest distance significantly reduced unnecessary computations. However, DFS struggled with guaranteeing optimal solutions for more complex graphs due to its depth-first exploration strategy, which could lead to redundant paths being explored before backtracking.

On the other hand, BFS, designed with an iterative queue-based approach, consistently found the shortest path quicker in almost all scenarios. Its level-wise exploration ensured optimal results, particularly for simpler routes or graphs with direct connections. Despite its reliable accuracy, BFS required more memory than DFS, as it stored multiple paths simultaneously. This limitation became evident when there was no optimization. In these cases, the BFS algorithm could not even run without causing heap overflow.

Both algorithms demonstrated improved performance when enhanced with pruning mechanisms. These optimizations minimized the exploration of unnecessary paths, therefore reducing runtime and improving efficiency. However, we couldn’t uncover the patterns when which of the algorithms got results faster.

In conclusion, the choice between BFS and DFS for solving the shortest path problem depends on the specific constraints and requirements of the application. BFS performs better for scenarios where memory availability is not a concern, and accuracy in finding the shortest path is critical. DFS, with its lower memory usage, is suitable for environments where memory resources are limited. Both algorithms, when tailored to the problem context, offer robust solutions to the shortest path problem in weighted graphs.

## Labor Authentication:

| Task | Group Member Responsible |
|---|---|
| `Node, SinglyLinkedList, LinkedStack, LinkedQueue` Class Implementations | Cem Alp Özer & Tolga Neslioğlu |
| `ArrayList` Class Implementation | Cem Alp Özer |
| `CreateArray` Class Implementation (CSV Reader) | Tolga Neslioğlu |
| DFS Algorithm Implementation | Tolga Neslioğlu |
| BFS Algorithm Implementation | Cem Alp Özer |
| Troubleshooting, Optimization, Error Detection and Fixes on Codes | Cem Alp Özer & Tolga Neslioğlu |
| GUI implementation (MyGUI Class) and creating executable JARs | Cem Alp Özer |
| Custom Class Time Complexity Analysis | Efe Ersöz & Yağmur Kaykaç & Beyza Zerdalı & Tolga Neslioğlu & Cem Alp Özer |
| DFS and BFS Time Complexity Analysis | Cem Alp Özer & Tolga Neslioğlu |
| Report Preparation | Efe Ersöz & Yağmur Kaykaç & Beyza Zerdalı & Tolga Neslioğlu & Cem Alp Özer |
| Presentation File | Yağmur Kaykaç |