COMP201 - Data Structures and Algorithms
Final Project Report

Project Name: SHORTEST PATH PROBLEM

Nurbanu KARAKAYA 042201059- Computer Engineering

Kerim ERCAN 042401167 - Computer Engineering

Ata SARGULLAR 042401175 - Computer Engineering

Arda Tuna KÜPÇÜ 042301093 - Computer Engineering

Instructor: Yassine Drias

Teaching Assistant: Mustafa Ersen

1. Introduction

1.1 Problem Statement

This project investigates the Shortest Path Problem through the application of two core graph traversal algorithms: **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. Determining the shortest and most efficient routes between multiple locations is a critical problem, influencing factors like cost and time. Cities can be represented as nodes (vertices), while the distances between them are modeled as weighted edges. The main challenge is to identify the optimal route between two cities while optimizing for specific objectives, such as reducing travel distance or minimizing execution time.

1.2 Project Objectives

The main goals of this project are outlined as follows:

1.Development of Graph Traversal Algorithms

Implement Depth-First Search (DFS) and Breadth-First Search (BFS) to identify paths between city pairs in a graph. This includes creating custom implementations of Stack and Queue data structures, without relying on pre-built Java utilities.

2.Algorithm Comparison

Analyze and compare DFS and BFS by examining their traversal techniques, efficiency, and practical applications. While both algorithms are essential for graph traversal, their behavior varies based on the graph's structure and the specific problem being addressed.

2. Algorithm and Data Structure Descriptions

2.1 Depth-First Search (DFS)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along a single path before backtracking to explore alternative paths. It utilizes a Stack data structure to manage visited nodes and the next nodes to explore. In this project, a custom Stack implementation was used to replicate this behavior.

Advantages of DFS	Disadvantages of DFS
DFS is straightforward to implement and effective for problems requiring exploration of all possible paths.	In unweighted graphs, DFS does not guarantee finding the shortest path, as it may explore longer routes first.
Its memory usage is efficient for graphs that are narrow and deep, as the stack size depends on the graph's depth.	For graphs that are very deep or contain cycles, DFS risks revisiting nodes unless a visited set is used.
	DFS is best suited for tasks where traversing all nodes is essential, and path length is not critical.

2.2 Breadth-First Search (BFS)

Breadth-First Search (BFS) traverses a graph layer by layer, beginning at the source node and visiting all of its immediate neighbors before proceeding to the neighbors of those nodes. It employs a Queue data structure to manage nodes, ensuring they are processed in the order they are added, following a first-in, first-out (FIFO) approach.

Advantages of BFS	Disadvantages of BFS
Guarantees finding the shortest path in unweighted graphs, making it ideal for pathfinding tasks.	Can be memory-intensive as it stores all nodes at the current level in the queue, especially for wide graphs.
Performs efficiently for shallow graphs or when the target node is close to the starting point.	Does not explore all possible paths, unlike DFS.

2.3 Differences Between Breadth-First Search (BFS) and Depth-First Search (DFS)

ASPECT	BFS (Breadth-First Search)	DFS (Depth-First Search)
Memory Usage	Requires more memory, especially for wide graphs, as all nodes at the current level must be stored in the queue.	Requires less memory for narrow and deep graphs since the stack size depends on the depth of the graph.
Data Structure Used	Queue (First-In-First-Out, FIFO)	Stack (Last-In-First-Out, LIFO)
Traversal Approach	Explores the graph level by level, visiting all neighbors of a node before moving to the next level.	Explores as far as possible along one branch before backtracking to explore other branches.
Shortest Path Guarantee	Guarantees finding the shortest path in unweighted graphs.	Does not guarantee finding the shortest path; may explore longer paths first.
Cycle Handling	Requires a visited set to avoid revisiting nodes.	Also requires a visited set to prevent infinite loops in cyclic graphs.
Exploration Depth	Limited to one level at a time.	Can explore deep paths before backtracking.
Use Case	Ideal for finding the shortest path or searching shallow graphs.	Suitable for exploring all paths or solving problems where path length is not critical.

2.4 Which Algorithm Should We Use?

Option for BFS when searching for the shortest path in an unweighted graph or if the target node is expected to be near the starting point. BFS is especially effective for tasks like route optimization, shortest path discovery, and traversing shallow graphs.

Choose DFS when the objective is to explore all potential paths, such as solving puzzles, identifying connected components, or analyzing narrow graphs. DFS is also advantageous for large but shallow graphs, as it consumes less memory compared to BFS.

While both DFS and BFS are built on similar principles, their behaviors and applications make them suitable for different types of problems. By evaluating their performance and results on a real-world graph of Turkish cities, we can clearly observe their respective strengths and weaknesses

3. Implementation Details

This project aims to implement two core graph traversal algorithms, Depth-First Search (DFS) and Breadth-First Search (BFS), utilizing custom-designed Stack and Queue data structures. This section outlines the program's structure, highlights key implementation decisions, and explains the reasoning behind them.

3.1 Java Program Structure

1.The graph is represented as an Adjacency List, implemented using a HashMap. In this structure, each key represents a city (node), while the corresponding value is a list of neighboring nodes along with their respective distances.

The Adjacency List was selected for its space efficiency, particularly for sparse graphs, which are typical in real-world scenarios like city-to-city connections. Unlike an adjacency matrix, it does not allocate memory for edges that do not exist, making it more efficient.

2.Custom Stack and Queue Implementations:

To fulfill the assignment requirements, custom implementations of Stack and Queue were created instead of using Java's built-in classes.

Custom Stack: Built using a **Linked List** with the following basic operations:

push(): Adds an element to the top of the stack.

pop(): Removes and returns the top element of the stack.

peek(): Retrieves the top element without removing it.

isEmpty(): Checks whether the stack is empty.

Custom Queue: Implemented using a **Linked List** with two pointers (front and rear) and the following operations:

enqueue(): Adds an element to the rear of the queue.

dequeue(): Removes and returns the front element of the queue.

peek(): Retrieves the front element without removing it.

isEmpty(): Checks whether the queue is empty.

Rationale:

A Linked List was chosen for both the Stack and Queue implementations because it allows for constant-time insertion and deletion operations (O(1)), which are essential for the efficiency of DFS and BFS.

Creating custom data structures showcases a solid understanding of fundamental data structures and avoids dependency on pre-built libraries.

3. **Graph Traversal Algorithms**: The program implements two main traversal algorithms:

- **Depth-First Search (DFS)**:

  -DFS uses the **Custom Stack** to explore nodes as deeply as possible before backtracking.

  -The algorithm marks visited nodes to avoid revisiting, ensuring the search terminates efficiently.

Breadth-First Search (BFS):

-BFS uses the **Custom Queue** to explore nodes level by level.

-Like DFS, BFS also marks nodes as visited to prevent redundant processing

4. **Path Construction**: Both DFS and BFS store parent nodes to reconstruct the final path from the start city to the goal city. A HashMap is used to keep track of each node's parent, which helps trace the path backwards once the goal is found.

Rationale:

Using a parent map simplifies path reconstruction without requiring recursion, which could increase memory usage for large graphs.

4. Complexity Analysis

This section provides a theoretical analysis of the time and space complexities of the implemented DFS and BFS algorithms, supplemented by empirical observations from the program's execution.

4.1 Time Complexity

Depth-First Search (DFS):

- **Theoretical Complexity**: O(V + E)
  - **V**: The number of vertices (cities).
  - **E**: The number of edges (connections).
- Each vertex is visited once, and each edge is traversed once during neighbor exploration.
- Operations on the custom stack (push() and pop()) have a constant time complexity of O(1).

Breadth-First Search (BFS):

- **Theoretical Complexity**: O(V + E)
  - Similar to DFS, BFS visits each vertex and processes each edge exactly once.
- Operations on the custom queue (enqueue() and dequeue()) also have a constant time complexity of O(1).

4.2 Space Complexity

DFS	
Space Used by Stack:	O(V)O(V)O(V) in the worst case, where all vertices are stored in the Stack.
Visited Set:	O(V)O(V)O(V), one entry for each vertex
Parent Map:	O(V)O(V)O(V), to store the parent of each vertex
Total Space Complexity:	O(V)O(V)O(V).

BFS	
Space Used by Stack:	O(V)O(V)O(V) in the worst case, where all vertices at a level are stored in the Queue.
Visited Set:	O(V)O(V)O(V).
Parent Map:	O(V)O(V)O(V).
Total Space Complexity:	O(V)O(V)O(V).

4.3 Empirical Observations

Practical Performance Analysis of DFS and BFS

To evaluate the practical performance of DFS and BFS, the program recorded execution times for various city pairs using a graph of Turkish cities. The key findings are as follows:

Shortest Path:

BFS reliably identified the shortest path due to its level-by-level traversal approach.

DFS, on the other hand, occasionally followed longer paths initially before backtracking to find shorter routes.

Execution Time:

BFS was faster when the target city was located closer to the starting city, as it processes nodes in layers.

DFS execution time varied based on the order of neighbors in the adjacency list and the overall depth of the graph.

Start City	Goal City	Algorithm	Path	Total Distance	Execution Time
Istanbul	İzmir	DFS	Istanbul → Denizli → İzmir	887 km	1.78 ms
Istanbul	İzmir	BFS	Istanbul → Denizli → İzmir	1040 km	0.97 ms

Insights:

-*BFS* reliably finds the shortest path but can require more memory when processing multiple neighbors at each level.

-*DFS* is more memory-efficient for narrow or deep graphs but may take longer to find an optimal path.

The comparison between DFS and BFS emphasizes their respective advantages and limitations in various scenarios. BFS excels in shortest-path problems, while DFS offers a straightforward solution for exhaustive searches. The use of custom Stack and Queue implementations ensured efficient algorithm performance with minimal overhead.

5.Results and Analysis

5.1 Path and Runtime Comparisons for DFS and BFS

- Experiments were performed on eight city pairs to assess the performance of the Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms. Visual representations of the paths and execution times are presented below.

DFS	
Path Comparison	**DFS paths varied significantly due to its deep-first nature, often resulting in longer paths.**
Runtime Comparison	**Execution time was inconsistent, highly dependent on the graph's depth and adjacency list order.**

BFS	
Path Comparison	**BFS consistently produced the shortest paths due to its level-by-level traversal**
Runtime Comparison	**Shallow graphs ran efficiently with much more predictable runtimes using BFS.**

Visual Comparisons

- Graphs and tables generated showing path lengths and times. The average runtime was lower as well as the path for BFS compared to DFS.

6. Discussion of Limitations and Improvements

Memory Usage:

-BFS consumed significant memory as it had to store all nodes at the current level during traversal.

-DFS sometimes generated suboptimal paths, demonstrating its limitations for shortest-path problems.

Algorithm Efficiency:

-DFS faced challenges with cyclic graphs, necessitating careful tracking of visited nodes to prevent infinite loops.

Potential Improvements:

Enhanced Data Structures:

-Incorporate advanced data structures like priority queues in BFS to further enhance performance.

Optimized Graph Representation:

-For denser graphs, utilize adjacency matrices to achieve faster lookups.

Bidirectional Search:

-Implement bidirectional BFS to minimize the search space and boost runtime efficiency.

Additional Features:

Visualization Tool:

-Adding a graphical interface to display real-time traversal would enhance user understanding and engagement.

Support for Weighted Graphs:

-The program could be expanded to handle weighted edges by integrating algorithms like Dijkstra's or A* algorithms.

Parallel Execution:

-Running BFS and DFS in parallel would enable direct comparisons and provide a more comprehensive performance evaluation.

-By incorporating these enhancements, the program could evolve into a robust utility for analyzing graph traversal algorithms across various scenarios.

7. **Conclusion**

This project centered on the implementation and analysis of two fundamental graph traversal algorithms, Depth-First Search (DFS) and Breadth-First Search (BFS). By utilizing custom data structures and applying these algorithms to a real-world graph of Turkish cities, the study offered valuable insights into their strengths and weaknesses in addressing various challenges, particularly the Shortest Path Problem.

Through its implementation, DFS proved to be an effective algorithm for tasks that require exploring all possible paths, such as solving puzzles, identifying connected components, or navigating narrow and deep graphs.

Its use of a custom Stack data structure allowed for efficient memory utilization, making it ideal for scenarios with memory limitations. However, a significant drawback of DFS is its inability to guarantee the shortest path in unweighted graphs, as it often delves into deeper, longer routes before backtracking to shorter ones. Additionally, its performance is heavily influenced by the graph's structure and the sequence of neighbors in the adjacency list.

The custom implementations of Stack and Queue demonstrated a solid understanding of low-level data structures while ensuring efficient algorithm execution.