Final Report
Finding shortest path in Graph

Amin Maneshi

Supervisors:
Dr. Holger Meyer
M.Sc. Florian Rose

September 2024

Contents

1 Motivation . . . 3
2 Introduction . . . 3
2.1 Types of databases . . . 3
2.2 Types of Graphs . . . 4
2.3 Graph Traversal Methods . . . 5
2.4 Structured Query Language . . . 8
2.5 PostgreSQL . . . 9
2.6 PostgreSQL Configuration and Its Impact on Performance . . . 9
2.7 Supporting Tools: pgRouting and NetworkX . . . 10
2.8 Performance Enhancements through Indexing and Adjacency Lists . . . 11
3 Implementation . . . 12
3.1 Datasets . . . 12
3.2 Common Table Expressions . . . 17
3.3 User-Defined Function . . . 19
3.4 System Specifications . . . 20
4 Simulation . . . 21
4.1 Gneutella . . . 21
4.2 Texas Road Network . . . 24
5 Comparing UDF and CTE . . . 28
5.1 Gnutella . . . 28
5.2 Texas Road Network . . . 29
6 Results of simulations . . . 33
7 Conclusion . . . 33

List of Figures

1 Depth-First Search . . . 5
2 Breadth-First Search . . . 6
3 Bidirectional Breadth-First Search [22] . . . 7
4 Distribution of shortest path lengths . . . 14
5 Comparison of Statistical Distributions Fitted . . . 14
6 Q-Q Plot Comparing to a Normal Distribution . . . 14
7 In-Degree Distribution of Gnutella . . . 15
8 Out-Degree Distribution of Gnutella . . . 15
9 CTE (without adjacency) of Gneutella . . . 21
10 CTE (with adjacency) of Gneutella . . . 22
11 Average runtime of CTE in Gnutella . . . 22
12 UDF (without adjacency) of Gneutella . . . 23
13 UDF (with adjacency) of Gneutella . . . 23
14 Average runtime of UDF in Gnutella . . . 24
15 CTE (without adjacency) of Texas Road Network . . . 25
16 CTE (with adjacency) of Texas Road Network . . . 25
17 Average runtime of CTE in Texas Road Network . . . 26
18 UDF (without adjacency) of Texas Road Network . . . 26
19 UDF (with adjacency) of Texas Road Network . . . 27
20 Average runtime of UDF in Texas Road Network . . . 27
21 UDF and CTE (without adjacency) of Gneutella . . . 28
22 UDF and CTE (with adjacency) of Gneutella . . . 29
23 UDF and CTE (without adjacency) of Texas Road Network . . . 30
24 UDF and CTE (with adjacency) of Texas Road Network . . . 30
25 High level of the Texas Road . . . 32

1 Motivation

In the present inter-connected world, efficient computation of the shortest path in complex networks has gained vital importance in many domains to optimize performance and gain the best decisions. The objective of the project is to apply shortest path algorithms on two real, different datasets: Gnutella peer-to-peer file sharing network and road network of Texas. The shortest path algorithms applied on the Gnutella network will facilitate resource sharing much faster, reduce latency, and improve resilience in general. They minimize traveling time, reduce congestion on Texas road networks, and determine the best route in emergency responses and logistics. In the present project, we would solve the shortest path problem using different approaches and algorithms and then analyze and compare them in detail for their effectiveness and performance. By applying the shortest path problem to these diverse contexts, this work intends to bring out the fact that the wide applicability and potential of path finding, when done efficiently, has a transformative effect in digital and physical networks.

2 Introduction

2.1 Types of databases

Relation Table-Based Datasets

A relational database is such a type of database system where data are stored as a collection of tables, with rows and columns. Each table or relation represents some particular type of entity; every row of a relation represents a record having some unique data, and each column represents an attribute concerning that entity. Relational databases are based on the relational model developed by E.F. One is the Codd in 1970, emphasizing structured data, pre-defined schemes where data integrity and consistency are assured through primary and foreign keys. Its structure will enable comprehensive data management and retrieval via SQL queries to operate on several tables: select, project, join, and aggregate.[11]

Relational databases have been commonly utilized for transactional enterprise applications, such as banking, inventory management, human resources, and customer relationship management, since they support ACID (Atomicity, Consistency, Isolation, Durability) properties. However, they may experience difficulties when dealing with highly interconnected, semi-structured, or unstructured data as efficiently as they do in scenarios where there is structured data with well-defined relationships. This can easily be due to the necessity to join different tables in complex operations, which usually lead to poor performance when data grows. [25]

Graph database

A graph database is a database system in which the relationships between the objects or entities are as important as the objects themselves. In a graph database, data is represented as a graph, the components of which include nodes and node/edge features or properties. This makes it an extremely powerful database when dealing with complex, semi-structured, or densely connected data. Graph databases allow fast querying, often responding in milliseconds due to their ability to traverse through connected nodes directly without needing complex join operations.[2]

Several enterprise-level applications for graph databases include communication, healthcare, retail, finance, social networking, and online media. They support graph data modeling and therefore allow for efficient CRUD operations using index-free adjacencies, which enable high-performance traversal across large datasets. Unlike relational databases that store data in tables and require costly joins for relationships, graph databases use a property graph model, where relationships are first-class citizens. This design allows graph databases to scale better for highly connected data and enables much faster query performance when exploring relationships between data points, making them ideal for applications such as social networks, recommendation engines, and fraud detection.[17]

2.2 Types of Graphs

Directed Graphs

In a directed graph, the edges have a sense so that they form some kind of asymmetric relation between the nodes. More precisely, each edge is defined in terms of a head and a tail: the tail node is the one from which the edge emanates, and the head node is the one toward which the edge points. This directionality is absolutely fundamental wherever the relation leading from the edges is itself inherently directionally dependent, such as in a hyperlink leading from one web page to another, or a one-way street in a city map.[18]

Undirected Graphs

An undirected graph does not have a well-defined direction for an edge; the relation between two nodes in it is symmetric. This would mean all the edges that are connecting two vertices are bidirectional, as it is with social networks and several other real-life examples. An undirected edge between two nodes can also be represented by a pair of directed edges that are inverses of each other; hence, an undirected graph is a special case of a directed graph.

Weighted Graphs

A weighted graph is a graph in which each edge is tagged with some numerical value, which is often referred to as weight. This conventionally refers to cost, distance, or time consumption in crossing that corresponding edge. Hence, it is the particular advantage of weighted graphs to enable the representation of solutions such as the shortest path where the weight total amount is supposed to be minimized. For example, it could model a transportation network, where the weights were the lengths of time taken to travel between places or the actual distances, and it would then find the quickest or shortest route between two locations.[29]

Unweighted Graphs

An unweighted graph is a graph where all edges are considered equal, meaning that no specific numerical values (weights) are assigned to the edges. In an unweighted graph, the primary concern is the number of edges in a path rather than their associated costs. This type of graph is useful when all connections are equivalent, such as determining the minimum number of steps required to navigate a network or solving problems where only the structure of connections matters, without regard to varying distances or costs.[29]

Connected Graph

A graph is considered connected if there is a path between every pair of nodes. This means that any two nodes can be traversed through a series of edges, ensuring complete accessibility within the network.[10]

Disconnected Graph

A graph is disconnected if there does not exist a path between at least one pair of nodes. A disconnected graph would, therefore, show isolated subgraphs or components wherein nodes in each subgraph are connected to each other but no connectivity exists between different subgraphs. This happens quite frequently in fragmented networks, where there are partitioned or failed sections that cannot communicate with other parts of a network.[10]

Multigraphs

Multigraphs are graphs in which any two nodes may be connected by more than one edge. This comes about in instances in which the nodes are connected via several relations. For instance, in the case of the Gnutella peer-to-peer file-sharing network, nodes are users, and edges are file-sharing connections. Multiple connections between the same users give rise to multiple edges. Knowing multigraphs can, therefore, be very important in some cases while calculating the shortest path since multiple edges may offer alternative routes.[18]

Hypergraphs

Hypergraphs are graphs for which edges can connect any number of nodes; these are sometimes called "hyperedges." While they are often represented simply as a bipartite network, they can be very useful in modeling complicated relationships in networks. In the case of the Texas road network, for example, the hypergraph represents intersections (nodes) connecting multiple roads (hyperedges). This approach may return the shortest path if it is assumed that all possible road connections are considered simultaneously at an intersection.[18]

Hypernodes

Another generalization of a graph is the hypernode graph, where the nodes are replaced by hypernodes, which may consist of a set of nodes and edges. This can become very useful in scenarios of nested or hierarchical networks. For example, in the Texas road network, a hypernode could be a city, that is, a set of intersections and roads that is within the larger state network. This nesting feature can allow the shortest path calculation to be simplified by treating complex subgraphs as a single entity; this will reduce the complexity of the overall network.[18]

2.3 Graph Traversal Methods

Depth-First Search

Depth-first search(DFS) is one of the principal graph search algorithms for graph traversal. It starts to traverse deeply along the first edge from node A, thus going through all nodes on this branch before backtracking. That is, the algorithm visits all edges leaving the current node before backtracking to the previous node until a node is reached that has at least one unexplored edge leaving it. This continues until all reachable nodes have been visited. The beauty of DFS lies in its going as deep as possible into the graph structure before backtracking—exposing the detailed node connections. Figure 1 The Following figure shows DFS visit pattern starting from Node A. It elaborates on how it explores. figure shows the number in which order the nodes are visited by Algorithm, thus showing way it traverses.[22]

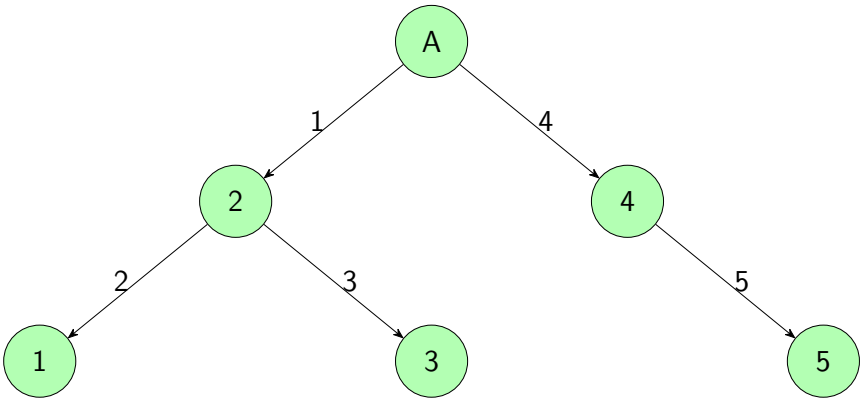

Figure 1: Depth-First Search

DFS is a recursive method where the concept of backtracking is used. It performs a full search of the nodes either in progressing ahead or backtracking. By definition, backtrack is moving ahead, a step back along the same path, when moving forward, and there is left no other node down the current path so that you can move further to find nodes to finally traverse all of them. All nodes will be visited in the current path until all of the unvisited nodes have been traversed after which the next path will be picked. This native recursive nature of DFS is pretty easy to implement by using stacks—the basic idea to select any node and start from it. Then, all its adjacent nodes get pushed to the stack. Choosing the next node to visit and pushing all its adjacent nodes into the stack is done by popping a node from the stack. So try to do this till the stack is empty. there is one condition: however, just be sure, that the nodes you are visiting are

marked. This will quite ensure that you are not visiting the same once-more node. If you do not mark the visited nodes, and you visit the same node more than once, you will end up in an infinite loop. The main drawbacks of DFS are the following. First, as it does not generally find the shortest path to the target node, DFS deepens as much as possible through each resource branch before backtracking. It may settle on longer paths before it has a chance to discover shorter ones. Second, recursive implementation of DFS may result in the stack overflow in deep charts, especially those with long paths and few branches, since every recursive call adds another frame to the stack; therefore, it may exceed the available stack limit. These limitations bring about the unsuitability of DFS in situations where pathfinding is optimal or deep recursion is likely to occur.[22]

Breadth-First Search

Breadth-First Search(BFS) (Figure 2) is a graph traversal algorithm that starts at one node and explores all other nodes at the present depth level before moving to nodes at the next depth level. That means the first level includes all neighbours of that source node. The next level, LEVEL i, includes all the neighbors of the nodes in the previous level, LEVEL i-1, that have not been visited in previous levels. The algorithm stops as soon as a path to the target node is found; it is not necessary that it attains a given maximum depth. Finally, BFS explores the graph as broadly as possible; that is, it goes to lower and lower depth levels, but only after the visit of all nodes at the current depth level is complete.[22]

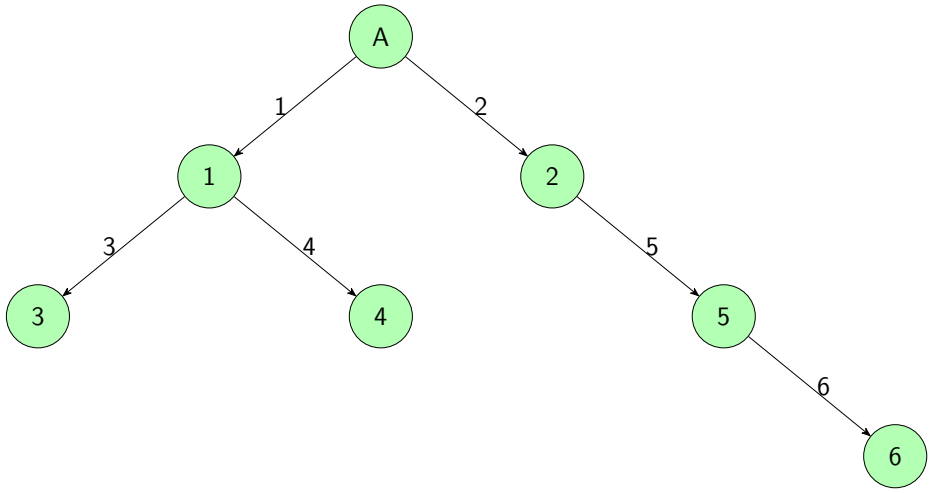

Figure 2: Breadth-First Search

It can let multiple paths to the same node be explored since it realizes BFS with the purpose of discovering all possible paths. This can give rise to revisiting nodes in case there exist more than one edge leading to them. This behavior will ensure an exhaustive search and guarantee the shortest path in an unweighted graph.[22]

Bidirectional Breadth-First Search

The bidirectional search algorithm is similar in nature to the unidirectional algorithm, but it makes one key efficiency: it performs two searches simultaneously—one forward from the start node and one backward from the end node. Termination conditions of the algorithm are twofold: All Nodes Visited: This occurs if there is no path from the start node to the end node. In such a case, both searches will terminate when there are no more nodes left to expand. In general, with bidirectional BFS, each search expands approximately half as many nodes as the corresponding unidirectional search. Meeting point found: If the searches meet, there is a path. The bidirectional BFS does not result in a shortest path always. The algorithm will then proceed until it has found a path of prespecified length, which again may be varied on the basis of graph or maze complexity.[22]

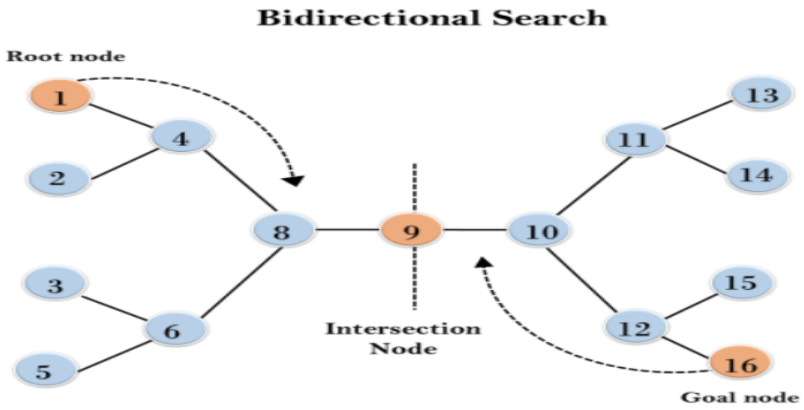

Figure 3: Bidirectional Breadth-First Search [22]

Dijkstra's Algorithm

Dijkstra's algorithm is a shortest-path graph traversal and the weighted graph algorithm. In a weighted graph, it finds the shortest path from a source vertex to all other vertices in a graph. In essence, the node with the smallest tentative distance (initialized to zero for the starting node, and to an infinitely large value for all others) is chosen, all its neighbors are explored, and their distances are updated if a new shortest way is found. This is done iteratively until all reachable nodes have their shortest path determined.

1. **Initialization**: Set the distance to the source node to zero and all other nodes to infinity.
2. **Priority Queue**: Use a priority queue (or min-heap) to keep track of nodes with the smallest tentative distance.
3. **Exploration**: At each step, remove the node with the smallest distance from the priority queue, update the distances to its neighboring nodes, and reinsert them into the priority queue if a shorter path is found.
4. **Termination**: The algorithm ends when all nodes have been visited, ensuring that the shortest path to each node has been found.

Dijkstra's algorithm is naturally greedy and best equipped to handle the case of non-negative weights of edges. As such, huge applications can be found in routing and analysis of networks, including the shortest distances in driving between places.[9].

A* Search Algorithm

A* extends Dijkstra by incorporating a heuristic to lead the exploration in finding the shortest path within the graph. This is very useful when the goal is to reach a particular target node rather than all nodes. The heuristic is responsible for the estimate of the cost from the current node to the target node, which will be helping the algorithm in prioritizing nodes that are most likely to lead to the shortest path.

1. **Initialization**: Similar to Dijkstra's Algorithm, initialize the cost of the starting node to zero and all others to infinity.
2. **Cost Function**: Maintain two costs for each node:
   - $\mathbf{g(n)}$: The known cost from the start node to the current node.
   - $\mathbf{h(n)}$: The heuristic cost, which estimates the shortest distance from the current node to the goal.
3. **Evaluation Function**: Calculate a score $\mathbf{f(n) = g(n) + h(n)}$. The algorithm explores nodes with the lowest value of $\mathbf{f(n)}$, where $\mathbf{h(n)}$ helps focus the search towards the target.

4. **Priority Queue**: Use a priority queue to select the node with the smallest **f(n)**.

5. **Exploration and Heuristic Guidance**: The algorithm explores the neighbors of the current node, updating their costs based on the evaluation function, and prioritizes the next node with the smallest combined score.

6. **Termination**: The algorithm stops once it reaches the target node and reconstructs the shortest path using backtracking.

The most important factor of A* would be the heuristic, which estimates the remaining cost to get the goal. E.g., on a grid, on pathfinding problems, h(n) could be a straight-line distance (Euclidean distance) from the current node to the target.

A* is both optimal and complete, that is, it is guaranteed to find the shortest path if the heuristic is admissible—it never overestimates the true cost—and consistent—it satisfies the triangle inequality.[14].

2.4 Structured Query Language

Structured Query Language (SQL), will be applied in this project for managing, querying, and analyzing huge amounts of datasets, including the Gnutella peer-to-peer network and the Texas road network. The main advantages of this technology include the following, which are going to be very helpful in this project:

1. Efficient Data Storage and Retrieval

The datasets used in this project, like Texas road networks, entail huge numbers of nodes and edges. SQL databases can effectively store large graphs. SQL provides the support mechanism for efficient and optimized queries on the data: it helps in retrieving the specific nodes, edges, or subgraphs without having to load the whole dataset into memory.[12].

2. Graph Representation and Querying

Graphs could be represented in SQL by tables that store nodes and edges. For instance, every node would be a row of some "nodes" table and all edges would be rows of an "edges" table with foreign keys linking the respective nodes. That would make a number of relational queries very easy such as retrieval of all neighbours of a node, or extraction of all edges linked to some node by mere execution of SQL queries over the database.[11].

3. Use of SQL for Graph Traversal and Pathfinding

In the project are implemented algorithms BFS, DFS, and Bidirectional BFS to perform graph traversals and to find paths between nodes. SQL supports recursive queries such as the `WITH RECURSIVE` clause, which can be used to implement these traversal algorithms. It follows that the graph traversals can directly be executed in the database, reducing the computation outside, hence increasing the processing speed for massive data sets.[12].

4. Data Preprocessing and Aggregation

In addition, SQL is used for purposes such as preprocessing and aggregation of data. Filtration of unnecessary nodes, or edges, aggregation of multiple connections, and preparation of the graph for traversal algorithms can easily be done through the use of SQL queries. This flexibility is going to reduce the complexity of data manipulation; clean and structure a dataset before the application of the graph traversal algorithms.[12].

5. Integration with Graph Algorithms

SQL Databases can be integrated with graph algorithms written in other programming languages. Run SQL queries extracting a useful part of the graph and feed it into a custom BFS, DFS, and Bidirectional BFS algorithm (among others) so as to leverage the power from both SQL-databases (in data storage and retrieval) and algorithms from outside for complex processing of the graph.[12].

SQL, in this project, supports the management of large-scale graph data much better. This would help store, retrieve, preprocess, and query the dataset by abstracting their application into graph traversal algorithms such as BFS, DFS, and Bidirectional BFS, which should be very efficient and scalable. SQL is very useful for the processing and handling of large-sized data in the form that is desirable for the graph-based analysis of this type.

2.5 PostgreSQL

In the project, PostgreSQL-a free and open source object-relational database system-is used in managing and analyzing Gnutella network and Texas road network datasets. Among others, PostgreSQL provides several important features and advantages which are particularly useful for large-scale graph-based analysis:

1. Advanced SQL Support

PostgreSQL offers comprehensive and standards-compliant SQL capabilities, making it ideal for complex queries required in graph traversal and analysis. Its support for advanced SQL features such as recursive queries (`WITH RECURSIVE`) allows efficient implementation of algorithms like BFS, DFS, and Bidirectional BFS directly within the database, reducing the need for external computation[20].

2. Performance and Scalability

PostgreSQL is known for its ability to handle large datasets efficiently. For this project, where the Texas road network and the Gnutella network involve thousands of nodes and edges, PostgreSQL provides the performance and scalability necessary to manage such data volumes. Its robust indexing mechanisms, including B-trees and hash indexes, help optimize query performance, enabling faster retrieval and processing of graph data[20].

3. Extensibility and Support for Graph Data Types

One of PostgreSQL's key strengths is its extensibility. It allows users to define custom data types and functions, making it particularly useful for graph-based applications. Additionally, extensions like `pgRouting` provide built-in support for graph algorithms, such as shortest path, driving distance, and route optimization. This makes PostgreSQL a powerful tool for managing and querying graph data, especially in road networks[20].

4. Reliability and ACID Compliance

PostgreSQL is highly reliable and provides full ACID (Atomicity, Consistency, Isolation, Durability) compliance, ensuring data integrity even during complex transactions. This is critical when running graph algorithms that involve multiple queries and updates, as it guarantees that the data remains consistent and error-free[20].

5. Open-Source and Active Community Support

PostgreSQL is open-source and has an active and supportive community. Its open-source nature means it is cost-effective for academic and research projects, while the active community ensures continuous updates, security patches, and new features. This makes it a reliable choice for long-term projects where database stability and security are essential[20].

PostgreSQL is used in this project because of its advanced SQL capabilities, performance, extensibility, and reliability. Its ability to efficiently handle large datasets and support complex graph queries makes it an excellent choice for analyzing the Gnutella and Texas road networks. The availability of graph-specific extensions further enhances PostgreSQL's suitability for this type of graph analysis.

2.6 PostgreSQL Configuration and Its Impact on Performance

To optimize the performance of database queries and simulations in this project, several PostgreSQL configuration parameters were adjusted in accordance with the system specifications. Below, we explain the effect of each configuration parameter and provide the values set for the project, based on the system's capabilities and default PostgreSQL settings.

- **work_mem:** This parameter determines the amount of memory allocated for each sort operation or hash table before spilling to disk. Increasing this value can significantly improve the performance of

operations that require sorting or hashing, especially when handling large datasets. For the project, `work_mem` was set to `1250 MB`, utilizing the available RAM to balance memory usage effectively without causing excessive memory pressure.

- **maintenance_work_mem:** This setting controls the amount of memory used for maintenance operations like `VACUUM`, `CREATE INDEX`, and `ALTER TABLE`. A higher value speeds up these operations but should be set carefully to avoid using too much memory during multiple simultaneous maintenance tasks. In the default setup, PostgreSQL allocates `64 MB`.
- **temp_buffers:** This parameter sets the amount of memory reserved for temporary buffers per session. Temporary buffers are used for operations such as sorting and storing intermediate results of complex queries. The default value is `8 MB`, but increasing this to `64 MB` can reduce disk I/O for temporary data handling, which is beneficial when dealing with large datasets as in this project.
- **effective_cache_size:** This parameter estimates the amount of memory available for disk caching by the operating system and PostgreSQL. It is not allocated but used by the query planner to decide how much data can be cached, influencing the execution plan of queries. The default is typically `4 GB`, making it optimal for query performance without overestimating cache capacity.
- **max_parallel_workers_per_gather:** This setting determines the number of workers that can be used for parallel query operations. On systems with multiple cores, increasing this number can significantly speed up query execution by parallelizing tasks. For a system with 12 logical processors, setting this value between `4` to `6` can balance between parallelism and resource contention.
- **parallel_setup_cost:** This parameter represents the estimated cost of launching parallel worker processes. A lower value makes the query planner more likely to use parallel execution plans. The default value is `1000`, which is reasonable for most systems. Adjusting it lower might slightly improve performance for smaller queries, but it is generally best left at its default unless fine-tuning for highly parallelized workloads.

2.7 Supporting Tools: pgRouting and NetworkX

In addition to using PostgreSQL for managing and querying large-scale graph datasets, this project also leverages **pgRouting** and **NetworkX** to verify the accuracy of the graph traversal algorithms and pathfinding results. These tools provide robust libraries and functions specifically tailored for graph operations and analysis.

pgRouting

pgRouting is an extension of PostgreSQL that adds support for geographic and graph-based algorithms. It is particularly well-suited for applications involving road networks, making it an ideal tool for this project's work with the Texas road network. pgRouting provides a wide range of built-in functions for graph algorithms, including:

- Shortest path algorithms
- Driving distance calculations
- Traveling salesman problem (TSP) solvers
- Route optimization functions

In this project, pgRouting is used as a reference to verify the results of custom implementations of algorithms like BFS, DFS, and Bidirectional BFS. By comparing the output from pgRouting's built-in functions with the results obtained from custom SQL-based queries, we ensure that the implemented algorithms are functioning correctly and efficiently[30].

NetworkX

NetworkX is a powerful Python library designed for the creation, manipulation, and study of complex networks (graphs). It supports both standard and advanced graph algorithms, and is widely used in the academic and research communities for network analysis. NetworkX provides a comprehensive set of tools for working with both weighted and unweighted graphs, including:

- Implementation of graph traversal algorithms
- Shortest path algorithms
- Tools for analyzing the structure and properties of graphs, such as centrality measures, connected components, and more

In this project, NetworkX is employed to cross-validate the custom graph traversal algorithms. After executing the algorithms on the Gnutella and Texas road network datasets, the results are compared with the output from NetworkX's implementation of the same algorithms. This double-checking ensures that the algorithms are correctly implemented and produce accurate results across both platforms[13].

Why pgRouting and NetworkX are Used in this Project

Both pgRouting and NetworkX are used as complementary tools to verify the correctness and efficiency of the graph traversal algorithms implemented in this project.

- **pgRouting** is particularly valuable for spatial networks like the Texas road network, where it simplifies complex routing tasks and provides accurate results for shortest path calculations.
- **NetworkX** offers a highly flexible environment for analyzing both spatial and non-spatial networks, such as the Gnutella peer-to-peer network. Its implementations of various graph algorithms allow for easy comparison with the custom algorithms developed in SQL.

By using these tools as references, this project ensures the accuracy of its results and highlights the benefits of integrating database solutions with specialized graph processing libraries.

2.8 Performance Enhancements through Indexing and Adjacency Lists

In this project, two key techniques were employed to enhance the performance of graph traversal algorithms: **indexing** and the use of an **adjacency list representation**. These optimizations were critical for efficiently handling the large datasets, such as the Gnutella network and the Texas road network, where fast query execution and memory-efficient graph representation were essential to achieve scalable and effective graph processing.

Indexing

Indexing is a widely-used database optimization technique that significantly improves query performance by allowing quick access to specific rows within a table. In this project, indexes were created on critical columns, such as node identifiers and edge relationships, to optimize the retrieval of graph data during traversal algorithms like BFS, DFS, and Bidirectional BFS.

By indexing the node and edge tables, the graph traversal algorithms could efficiently locate neighboring nodes during execution, reducing the time complexity associated with full table scans. This optimization was particularly important for large graphs, such as the Texas road network, where query performance directly impacted the overall efficiency of the simulation[15].

Adjacency List Representation

The adjacency list is a memory-efficient way to represent graphs, where each node is associated with a list of its adjacent nodes. This structure is particularly well-suited for sparse graphs, like the Texas road network, where many nodes are not directly connected. By using adjacency lists, the project minimized memory usage while allowing algorithms such as BFS and DFS to quickly retrieve neighboring nodes during traversal[4].

3 Implementation

As previously mentioned, our goal in this project is to find the shortest path in graph databases. In this section, we introduce the datasets used in this project and then examine two main approaches: User-Defined Functions (UDF) and Common Table Expressions (CTE).

3.1 Datasets

For this project, we utilized datasets from the SNAP (Stanford Network Analysis Project) repository, a widely recognized resource for large-scale network datasets. And the statistical metrics of each data set are taken from this site[19]

Gnutella

In this project, we analyze the Gnutella peer-to-peer file-sharing network dataset collected in August 2002 to study and implement algorithms for finding the shortest path in a graph. The dataset consists of 9 snapshots of the Gnutella network, where each snapshot represents a static image of the network at a given point in time.

Dataset Characteristics

- **Nodes**: Represent the hosts (computers) participating in the Gnutella network.
- **Edges**: Represent the communication links or file-sharing connections between the hosts.

Key Statistics

The dataset provides a set of useful metrics that offer insights into the network's structure:

- **Total Nodes**: 6,301
- **Total Edges**: 20,777

These metrics indicate the scale of the network, where each node has multiple connections with other nodes. In terms of shortest path analysis, having a large number of nodes and edges helps in testing the efficiency of algorithms like Bidirectional BFS, which can be used to compute the shortest communication path between two nodes in the network.

Weakly Connected Component (WCC)

- **Nodes in largest WCC**: 6,299 (99.97%)
- **Edges in largest WCC**: 20,776 (99.99%)

The largest weakly connected component (WCC) includes almost all the nodes and edges in the dataset, indicating that the majority of the network is connected, even if the directionality of the connections is ignored. This is significant because, in many graph algorithms, weakly connected components ensure that most nodes can be reached from one another, providing a strong test case for pathfinding algorithms [29].

Strongly Connected Component (SCC)

- **Nodes in largest SCC**: 2,068 (32.8% of nodes)
- **Edges in largest SCC**: 9,313 (44.8% of edges)

The largest strongly connected component (SCC) is the subgraph where every node is reachable from every other node through directed paths. In the context of shortest path algorithms, SCCs are essential because they represent the portion of the network where bidirectional communication is possible. Shortest path algorithms can only compute meaningful results within strongly connected components when directionality matters[27].

Clustering Coefficient

- **Average clustering coefficient**: 0.0109

The clustering coefficient measures the degree to which nodes in the graph tend to cluster together. In peer-to-peer networks like Gnutella, a low clustering coefficient (0.0109) suggests a sparse network where few nodes form tightly-knit groups. This impacts shortest path algorithms by indicating that the paths between nodes are likely to be longer, as there are fewer tightly clustered shortcuts available [28].

Number of Triangles

- **Number of triangles**: 2,383

A triangle in a graph represents three nodes that are all connected to each other. Triangles often indicate the presence of small, closed communities within the network. However, the Gnutella network has relatively few triangles compared to its size, further reinforcing the sparsity and decentralized nature of the network [21].

Fraction of Closed Triangles

- **Fraction of closed triangles**: 0.006983

This is the ratio of the number of closed triangles to the number of possible triangles. A fraction of 0.006983 suggests that only a small fraction of potential triangles are closed, meaning that the network has few tight-knit groups, which in turn affects the overall network structure and path lengths. This is typical for decentralized networks where connections between nodes are more ad-hoc and less structured[1].

Application to Shortest Path Analysis

The characteristics of the Gnutella network make it an excellent dataset for testing shortest path algorithms. The presence of both weakly and strongly connected components allows us to test the performance of various algorithms under different network conditions:

- **In the largest WCC**: We can apply shortest path algorithms without considering directionality, simulating situations where file-sharing links are bidirectional.
- **In the largest SCC**: We can focus on directed paths, testing the algorithm's ability to find the shortest route in a more constrained scenario.

Additionally, the low clustering coefficient and sparse triangle count indicate that the shortest paths are likely to span larger portions of the network, providing a more realistic test case for distributed algorithms like Bidirectional BFS. The real-world nature of the dataset ensures that the paths found are meaningful for decentralized network analysis.

- **Diameter (Longest Shortest Path):** The diameter of the graph is 20, which represents the longest shortest path between any two nodes in the network. This metric indicates the maximum number of edges that must be traversed to connect the most distant pair of nodes in the graph. A larger diameter suggests that the network is relatively sparse and that there are some nodes that are quite far apart from each other.
- **90-Percentile Effective Diameter:** The 90-percentile effective diameter of the graph is 6.5. This metric reflects the shortest path distance that covers 90% of all node pairs in the network. In other words, 90% of the node pairs can be connected through a path of 6.5 edges or fewer. This is an important measure as it gives an understanding of how compact or efficiently connected the network is for the majority of node pairs, despite the presence of longer paths that increase the overall diameter.

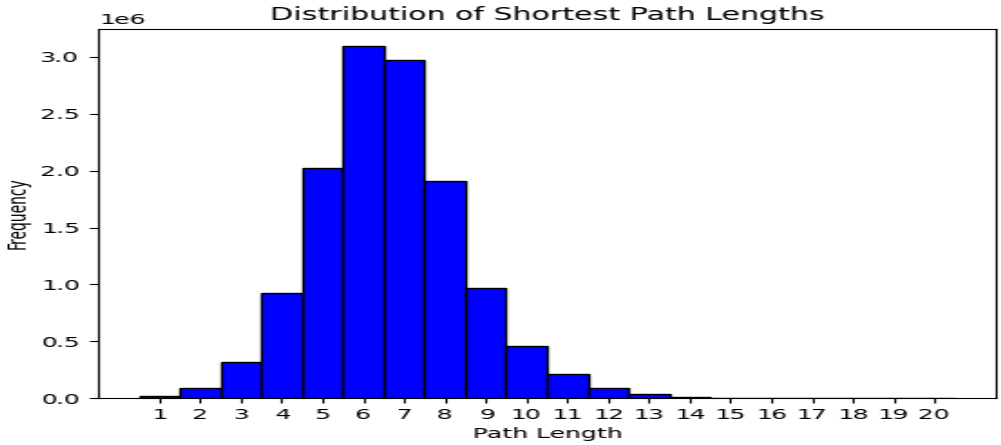

Figure 4: Distribution of shortest path lengths

Here (Figure 4) is a histogram of the distribution of shortest path lengths between nodes that occurred in the Gnutella peer-to-peer file sharing network. The x-axis represents length of shortest path between nodes, from 1 to 20, and the y-axis represents frequency. This distribution is a very clear bell curve centered around an approximate path length of 6-7, with the greatest frequency of the shortest path lengths of length 6 or 7 hops between nodes. As one moves away from that center, the frequency drops, showing that fewer node pairs are connected by longer paths. This corresponds to paths of length 6 or 7, and it means that the majority of the nodes of the Gnutella network are interconnected by relatively short paths-a typical feature of small-world networks.

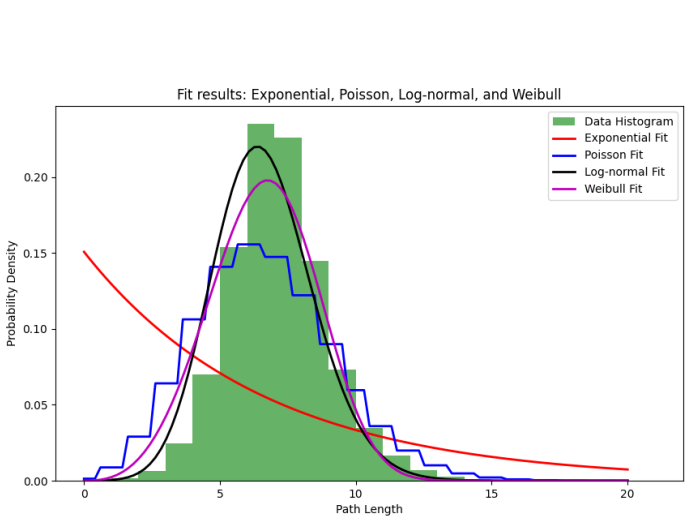

Figure 5: Comparison of Statistical Distributions Fitted

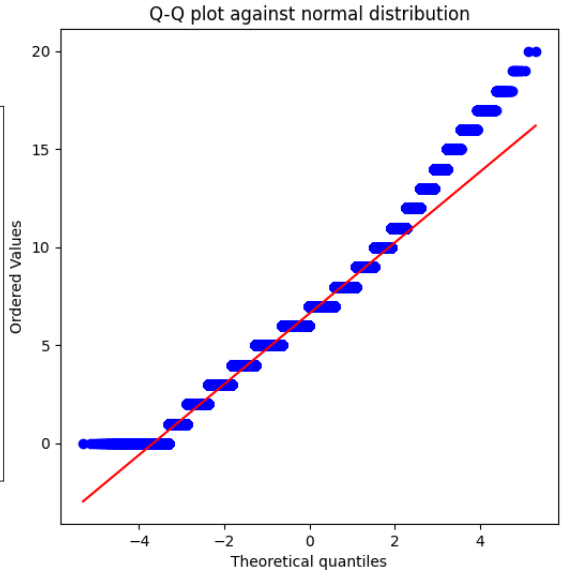

Figure 6: Q-Q Plot Comparing to a Normal Distribution

This graph (Figure 5) compares statistical distributions fitted to the Gnutella shortest path lengths. The log-normal fit (black) matches the data best, capturing both the peak and the tail, while the Poisson (blue) and Weibull (purple) fits are reasonable but less accurate. The exponential fit (red) poorly represents the data, especially for longer paths. The Q-Q plot (Figure 6) further confirms that the data does not follow a normal distribution, as the points deviate significantly from the red line, particularly at the tails, indicating a skewed distribution with heavier tails, consistent with the log-normal fit being the most appropriate.

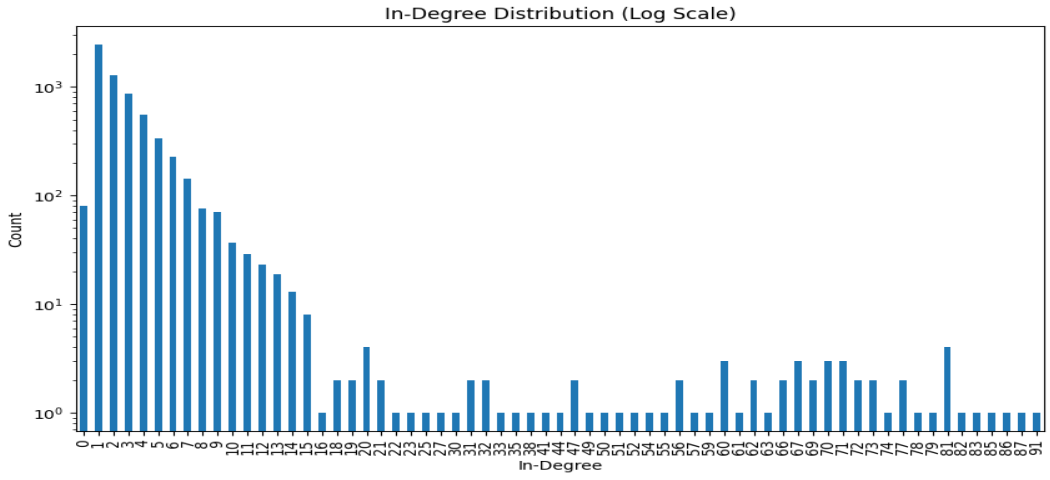

Figure 7: In-Degree Distribution of Gnutella

The in-degree distribution (Figure 7) shows that most nodes in the Gnutella network have few incoming connections, while a small number of nodes act as highly connected hubs. This pattern is important for shortest path algorithms because these hubs can significantly reduce the distance between other nodes, serving as key points for minimizing path lengths across the network.

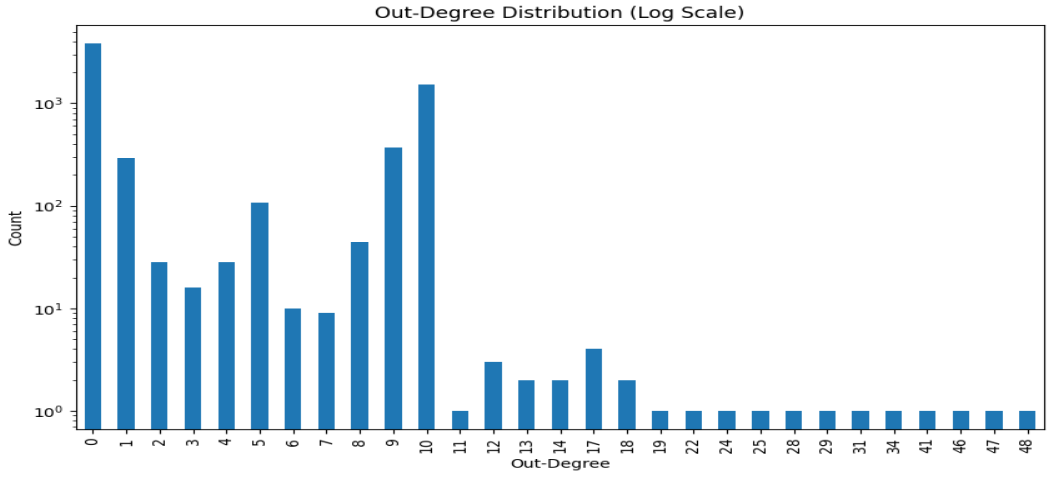

Figure 8: Out-Degree Distribution of Gnutella

The out-degree distribution (figure 8) indicates that most nodes have few outgoing connections, while some are highly active with many outgoing links. These high out-degree nodes are crucial for shortest path algorithms, as they provide multiple potential routes, helping to reduce the path length and improve the efficiency of navigation through the network.

Texas Road Network Dataset

This dataset represents the road network of Texas, where intersections and endpoints are modeled as nodes, and the roads connecting them are modeled as undirected edges. The dataset provides an ideal framework for studying shortest path algorithms due to its real-world application in transportation networks.

Dataset Characteristics

- **Nodes**: 1,379,917 (represent intersections and endpoints).
- **Edges**: 1,921,660 (represent undirected roads connecting intersections or endpoints).

This dataset is well-suited for shortest path analysis, as it represents a highly connected real-world network where roads (edges) connect intersections (nodes), enabling traversal across the entire state of Texas.

Largest Weakly Connected Component (WCC)

- **Nodes in largest WCC**: 1,351,137 (97.9% of total nodes).
- **Edges in largest WCC**: 1,879,201 (97.8% of total edges).

The largest weakly connected component (WCC) includes 97.9% of the total nodes, meaning almost all intersections in the Texas road network are part of a large connected subgraph, even when directionality is ignored. This indicates that most intersections are reachable from one another in some way, making the network suitable for comprehensive shortest path calculations. [23]

Largest Strongly Connected Component (SCC)

- **Nodes in largest SCC**: 1,351,137 (97.9% of total nodes).
- **Edges in largest SCC**: 1,879,201 (97.8% of total edges).

In undirected graphs, the weakly and strongly connected components are the same, meaning the largest SCC is also 97.9% of the total network. Every node in the largest SCC is mutually reachable, which is critical for algorithms that need bidirectional traversal, such as Bidirectional BFS . [27]

Clustering Coefficient

- **Average clustering coefficient**: 0.0470

The clustering coefficient measures the likelihood that a node's neighbors are also connected to each other. In this dataset, the clustering coefficient is relatively low (0.0470), indicating that local clustering of intersections is uncommon. [28]

Triangles

- **Number of triangles**: 82,869

The presence of 82,869 triangles indicates that there are small, tightly connected clusters in the network, though they are not very frequent relative to the network's size. [21]

Fraction of Closed Triangles

- **Fraction of closed triangles**: 0.02091

A fraction of 0.02091 suggests that a small portion of the possible triangles are actually closed, reinforcing the observation that local clustering is relatively low in the Texas road network. [1]

Diameter and Effective Diameter

- **Diameter (longest shortest path)**: 1,054
- **90-percentile effective diameter**: 670

The diameter of the network is 1,054, meaning the longest shortest path between any two nodes requires traversing 1,054 edges. This reflects the large geographic coverage of the Texas road network. The 90-percentile effective diameter is 670, indicating that 90% of the node pairs in the network can be connected by a shortest path that is at most 670 edges long, which suggests relatively efficient connectivity across the network. [21]

Handling the Texas Road Network Dataset

A Texas road network dataset with over 1,370,000 nodes and 1,920,000 edges presented a considerable computational challenge. It was impossible to create in-degree, out-degree, and shortest path length distribution plots and fit statistical models such as exponential, Poisson, log-normal, and Weibull distributions because of memory and processing time limits. These limitations precluded the realization of these analyses with the same level of detail as for the Gnutella network, but this limitation underlines very well the importance of the optimization of algorithms for large-scale graph analysis and points out that in the case of such data sets, one needs much stronger computational resources for future analysis.

3.2 Common Table Expressions

Introduction

Common Table Expressions (CTEs) are a powerful feature introduced in SQL Server 2005 and supported by later versions, including SQL Server 2008 and PostgreSQL 8.4 Development Version [26]. CTEs offer a more readable form of derived tables and can be declared once and referenced multiple times within a query. They also support recursion, allowing a recursive entity to be enumerated without the need for recursive-stored procedures [16].

General Description

A CTE is defined using the `WITH` statement and comprises an expression name, an optional column list, and a query. After definition, a CTE can be referenced like a table or a view in `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements. It can also be used in a `CREATE VIEW` statement as part of its defining `SELECT` statement [16].

The basic syntax for a CTE is:

```
WITH expr_name [(column_name [,...n])] AS
(CTE_query_definition)
```

Advantages and Disadvantages

CTEs offer several advantages over views and derived tables:

- **Readability:** CTEs improve query readability by breaking down complex queries into simpler, logical blocks.
- **Recursion:** CTEs support recursive queries, which are essential for hierarchical data retrieval [26].
- **Temporary Scope:** Unlike views, CTEs are temporary and exist only within the scope of a single SQL statement [16].

However, CTEs also have some limitations:

- **Keyword Restrictions:** CTE members cannot contain certain keywords such as `DISTINCT`, `GROUP BY`, `HAVING`, and `TOP`, which limits the complexity of queries [26].

- **Recursion Limitations:** Recursive CTEs are limited in that the recursive member can refer to the CTE only once [16].
- **Performance:** For large datasets, recursive CTEs may perform worse than equivalent queries using cursors or temporary tables [6].

CTE versus Temporary Tables

Temporary tables are session-specific tables created on disk and visible only within the session or stored procedure that created them. They can cause performance issues due to disk I/O and locking. CTEs, on the other hand, are more convenient as they do not require explicit creation and can be called immediately after definition [26].

CTE versus Derived Tables

Derived tables are subqueries in the `FROM` clause, which can only be used once per query and cannot be referenced by name outside of the statement where they are defined. CTEs improve upon derived tables by allowing multiple references within the same query, enhancing manageability and readability [16].

Recursion with CTEs

One of the most significant features of CTEs is their ability to support recursive queries. A recursive CTE is defined with an anchor member (base case) and a recursive member, which references the CTE itself. Recursion continues until no more rows are returned from the recursive member [8].

The basic structure of a recursive CTE is:

```
WITH RecursiveCTE AS (
    -- Anchor member
    SELECT ...
    UNION ALL
    -- Recursive member
    SELECT ...
    FROM RecursiveCTE
    WHERE ...
)
SELECT * FROM RecursiveCTE;
```

Guidelines for recursive CTEs:

- Must contain both an anchor and a recursive member.
- Both members must return the same number of columns with matching data types [7].
- Recursive CTEs can reference variables declared within the same batch or parameters used within the same code module [5].

3.3 User-Defined Function

In this project, **User-Defined Functions (UDFs)** were utilized as one of the approaches, alongside CTEs. A **UDF** is a custom function defined within a database to perform complex operations or calculations. Unlike built-in functions provided by the database system, UDFs allow users to encapsulate their own logic, computations, or processing steps that can be reused across multiple queries.

Types of UDFs

UDFs can be categorized into two primary types [3]:

- **Scalar UDFs**: These functions take input parameters and return a single value. For example, a scalar UDF might calculate a custom metric or perform specific data transformations, such as normalizing values or formatting strings.
- **Table-Valued UDFs**: These functions return a set of rows (a table). Table-valued UDFs are used to encapsulate complex query logic that results in multiple rows and columns, such as filtering or joining datasets in a reusable manner.

How UDFs Work

UDFs work similarly to traditional programming functions. You define a function by specifying its name, input parameters, return type (either scalar or table), and the function's body, which contains the logic to be executed. Once defined, UDFs can be invoked within SQL queries just like built-in functions. For instance, a scalar UDF in PostgreSQL might be defined as follows:

```
CREATE FUNCTION calculate_discount(price NUMERIC, discount_rate NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN price * (1 - discount_rate);
END;
$$ LANGUAGE plpgsql;
```

This function can then be used in queries:

```
SELECT calculate_discount(price, 0.15) AS discounted_price
FROM products;
```

Advantages of UDFs

The use of UDFs provides several advantages:

- **Reusability**: UDFs allow encapsulation of commonly used logic into a single, reusable function, eliminating the need for repeated complex expressions in multiple queries.
- **Modularity**: They promote code modularity and organization, making SQL queries more readable and maintainable by abstracting complex logic into a function.
- **Flexibility**: UDFs provide flexibility, allowing users to implement business rules or logic that may not be easily expressed with built-in SQL functions.
- **Abstraction**: By abstracting logic, UDFs ensure that users do not need to rewrite the same logic across different queries.

Performance Considerations

Although UDFs offer significant benefits, they can sometimes introduce performance overhead, particularly if they are not optimized or called excessively within large queries. This occurs because each UDF invocation may introduce additional computation time, especially when processing large datasets. Some databases might face optimization limitations with UDFs, such as reduced parallelism or inability to push down certain optimizations within the function's logic.

Thus, it's crucial to balance the use of UDFs with considerations for query performance, particularly in high-throughput scenarios. [24]

UDFs in Shortest Path Algorithms

In this project, the datasets used—Gnutella and Texas road network—are **unweighted**. As such, the shortest path algorithms applied were **BFS**, **DFS**, and **Bidirectional BFS**, which are well-suited for unweighted graphs.

UDFs can still play a valuable role even in unweighted graphs. For example, in BFS or Bidirectional BFS, a UDF can encapsulate the logic for expanding nodes, managing queues, or determining neighbors for traversal. Although the edge weights are uniform, UDFs help modularize the logic and make the pathfinding algorithms more efficient and reusable across various graph datasets.

BFS explores the graph level by level, expanding all neighbors of a node before moving on to the next level. **DFS**, on the other hand, explores as far along a branch as possible before backtracking. **Bidirectional BFS** optimizes the search for the shortest path by simultaneously searching from both the start and goal nodes, meeting in the middle and drastically reducing the search space.

In these applications, UDFs are useful for handling repetitive operations within the graph traversal, allowing for cleaner, more organized SQL queries that execute BFS or DFS logic efficiently.

3.4 System Specifications

To perform the simulations and analyses in this project, the following system configuration was used:

- **Processor:**
- Model: Intel Core i7-9750H
- Clock Speed: 2.60 GHz
- Cores: 6
- Logical Processors: 12

- **Memory:**
- RAM: 16 GB
- Work Memory: 1250 MB
- Effective Cache Size: 4 GB
- Shared Buffer: 4 GB

- **PostgreSQL Configuration:**
- work_mem: 1250 MB
- maintenance_work_mem: 1 GB
- temp_buffers: 64 MB
- effective_cache_size: 4 GB
- max_parallel_workers_per_gather: 4
- parallel_setup_cost: 1000

4 Simulation

In this section, we evaluate the performance of the two approaches, UDF and CTE, as discussed in the previous section. It is important to note that in each approach, we have used three methods: BFS, DFS, and Bidirectional BFS, and we will analyze the performance of each. Additionally, we will examine the performance of the various methods used to improve performance, as previously mentioned.

Indexing

Given that significant performance improvements were observed in all simulations through the use of indexing techniques, we applied this technique to both datasets and in all methods .

```
-- Create an index on source_node_id
CREATE INDEX idx_source_node_id ON "Name_of_table"(source_node_id);

-- Create an index on destination_node_id
CREATE INDEX idx_destination_node_id ON "Name_of_table" (destination_node_id);
```

Adjacency List Representation

Given that the adjacency method in some cases shows similar or even weaker performance, we conducted all steps once using this technique and once without it. in table 1, you can see the small section of the table related to Texas Road that have been transformed into adjacency format.

source_node_id	adjacency_nodes
0	{1, 2, 29}
1	{0, 23, 32}
2	{0, 26, 34}
3	{4, 7, 40}
4	{3, 18, 19}

Table 1: Adjacency Table for source_node_id and adjacency_nodes

4.1 Gneutella

In this section, we first evaluate the performance of the different approaches mentioned earlier on the Gnutella dataset.To increase the accuracy of the experiment, the simulation at each depth level was repeated 30 times with random paths.

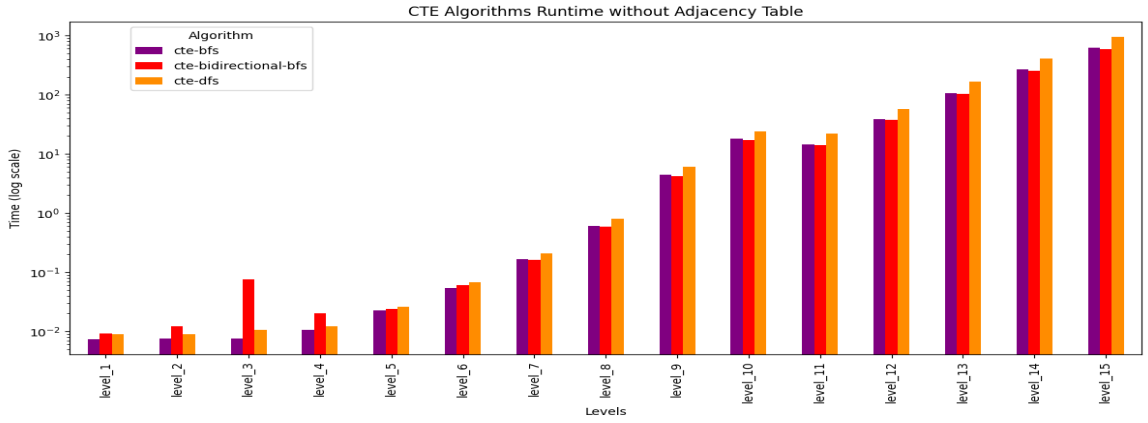

Figure 9: CTE (without adjacency) of Gneutella

In figure 9, each level represents the depth of each path, and at each depth, the experiment was repeated 20 times to increase accuracy, with the average results shown. As observed, up to depth 9, BFS generally performs better than the other algorithms. However, after depth 9, Bidirectional BFS starts to perform better. This can be attributed to the initial overhead of the Bidirectional BFS algorithm, which, due to its greater complexity, becomes more noticeable at lower depths where the shortest path search time is very short.

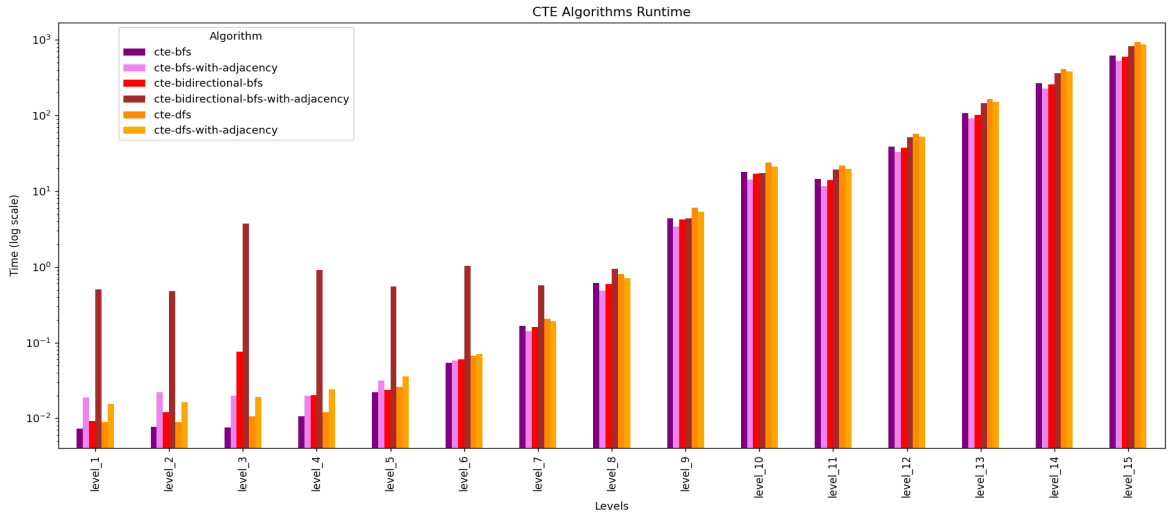

Figure 10: CTE (with adjacency) of Gneutella

In figure 10, we compare the algorithms with and without using the adjacency technique. As observed, up to depth 7, the overall performance of the algorithms without using adjacency was better. However, after depth 7, BFS with adjacency starts to outperform the normal version. The same holds true for the DFS algorithm. For the Bidirectional BFS algorithm, from depth 9 onward, the results are very close, with the performance sometimes being better and sometimes worse. It can be concluded that using this technique increases the initial overhead, as previously mentioned.

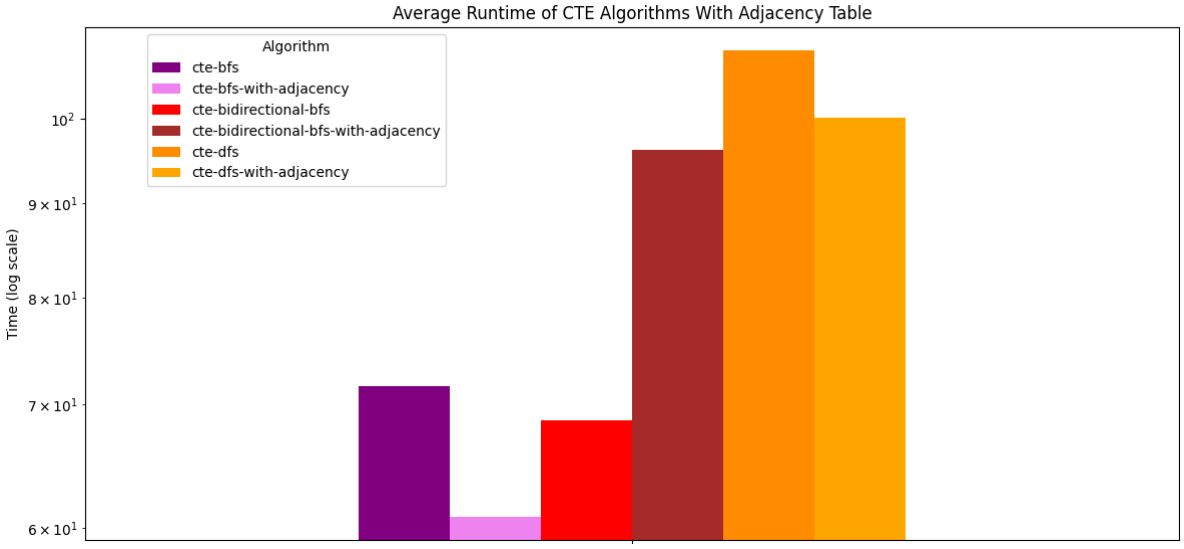

Figure 11: Average runtime of CTE in Gnutella

As shown in figure 11, we calculated the measured times for each algorithm separately over the course of all experimental stages. As is evident, BFS with adjacency exhibited the best performance. Additionally, DFS performance improved with the use of adjacency. However, this was not the case for Bidirectional BFS, where performance significantly deteriorated.

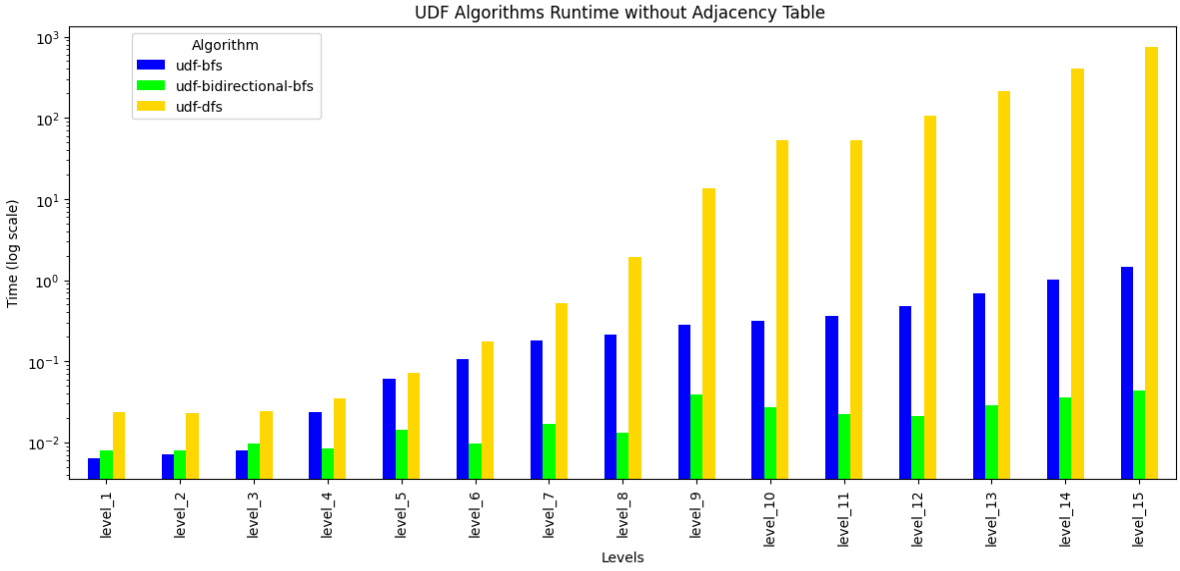

Figure 12: UDF (without adjacency) of Gneutella

In this graph(figure 12), we evaluated the UDF approach. As can be seen, initially, the BFS algorithm performs the best up to depth three. However, after that, Bidirectional BFS shows much better performance, which, as previously mentioned, is due to the high initial overhead of this algorithm. On the other hand, DFS consistently shows weaker performance, and as the depth increases, the difference becomes more significant.

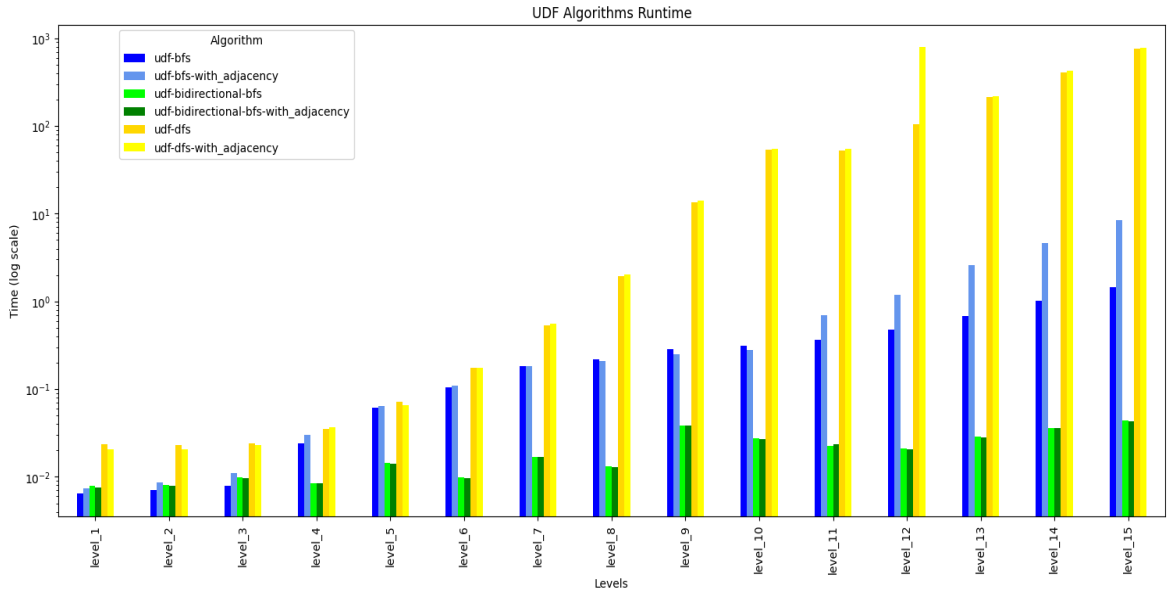

Figure 13: UDF (with adjacency) of Gneutella

As shown in figure 13, the use of the adjacency technique has not significantly improved performance overall. It can only be observed that in some depths, the adjacency technique performs slightly better than the normal case, with this improvement being more noticeable in the DFS algorithm. However, overall, it has not had a substantial impact, unlike the CTE approach.

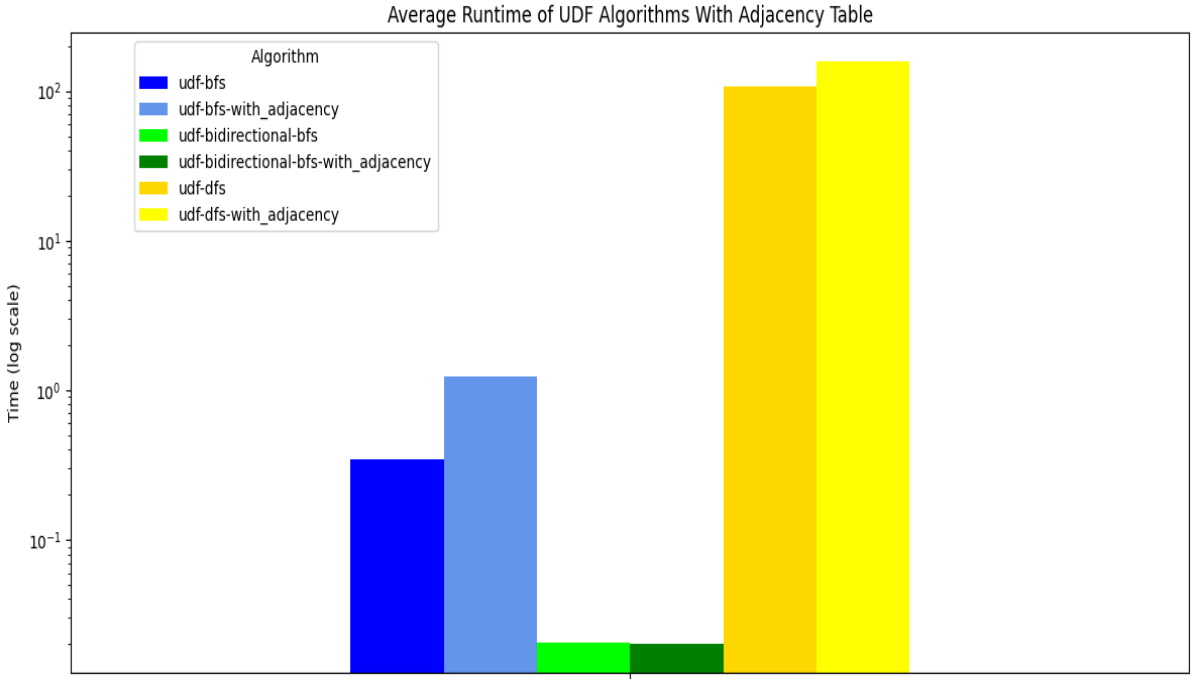

Figure 14: Average runtime of UDF in Gnutella

As was evident from figure 14, overall, the performance of the algorithms without using the adjacency technique was better, and this is more clearly shown in this plot. However, aside from this point, it should also be noted that unlike the CTE approach, in UDF, Bidirectional BFS demonstrated the best performance. As seen in the previous figures(figure 13), this difference becomes more pronounced as the depth increases, and the average computation time of this algorithm throughout the experiment was significantly lower than the other algorithms. It is worth mentioning that in the section comparing UDF and CTE, we will conduct a detailed comparison of the performance of these two approaches.

4.2 Texas Road Network

In this section, we will examine the Texas Road dataset, which is an undirected and large dataset. As in the previous section, we will first analyze the CTE approach, followed by the UDF approach. In both approaches, the performance will be assessed once with the adjacency technique and once without it. Additionally, each level in the charts represents the depth of the path considered in each measurement. To increase the accuracy of the experiment, the simulation at each depth level was repeated 30 times with random paths.

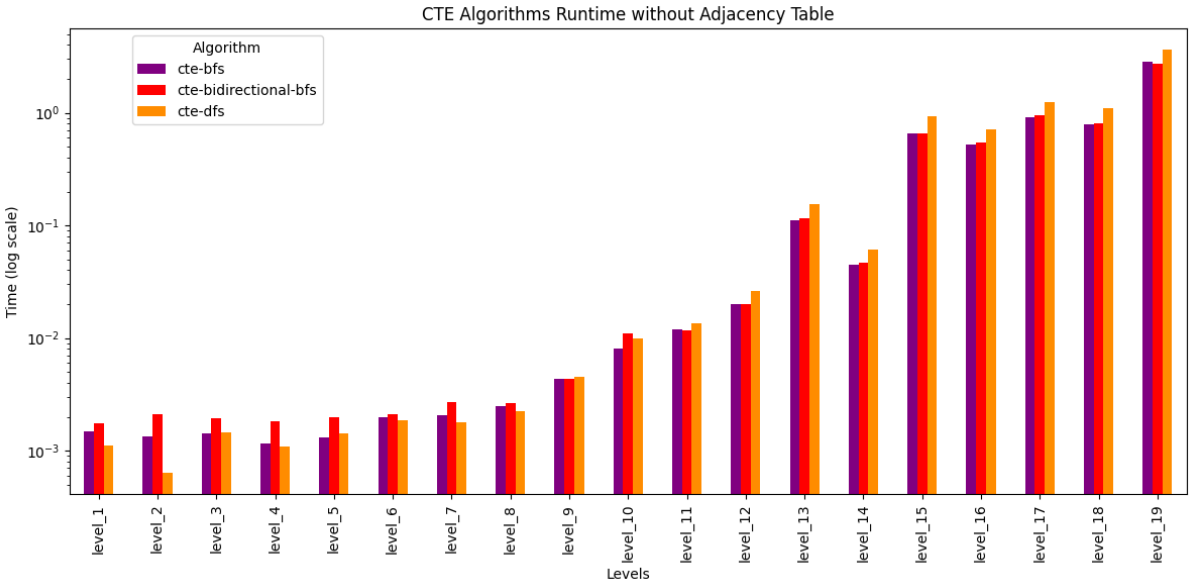

Figure 15: CTE (without adjacency) of Texas Road Network

As observed in Figure 15, the overall trend for all three methods—BFS, DFS, and Bidirectional BFS—is upward as the path depth increases. However, up to level 8, DFS outperforms the other methods. With increasing depth, BFS shows better performance, which continues up to depth 13. After that, until level 19, BFS and Bidirectional BFS demonstrate nearly identical performance. It is evident that in paths with shallow depths, Bidirectional BFS performs worse than the other methods due to its initial overhead.

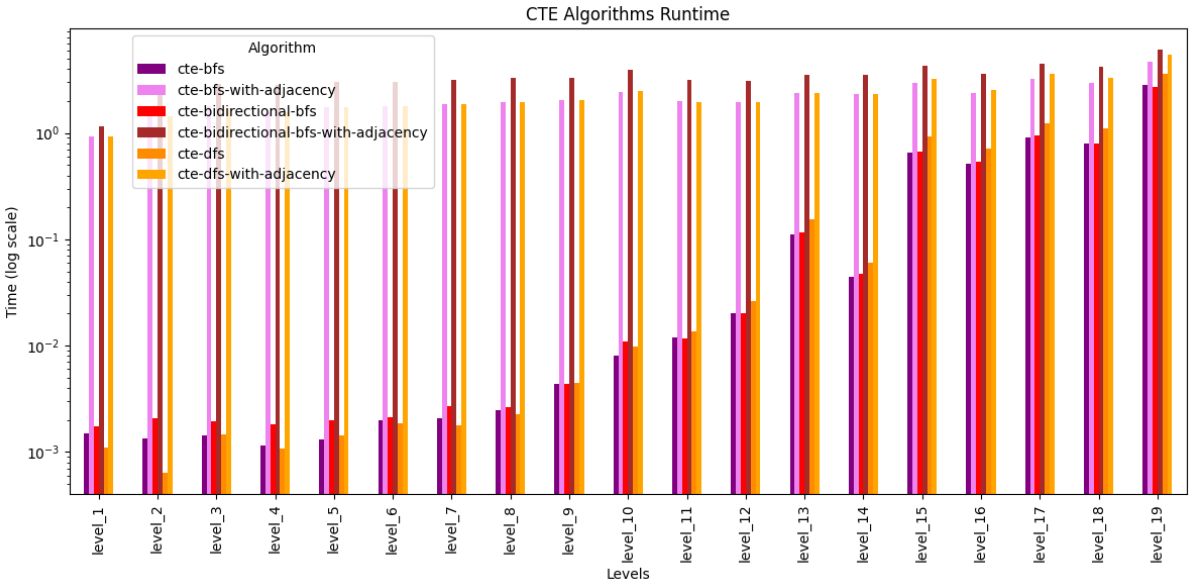

Figure 16: CTE (with adjacency) of Texas Road Network

Figure 16 demonstrates the use of the adjacency technique, which, as observed, does not improve performance. This is especially noticeable in paths with shallower depths, where the difference is quite significant. Overall, it can be concluded that this technique has not enhanced performance for this approach and dataset; in fact, a substantial decline in performance is evident.

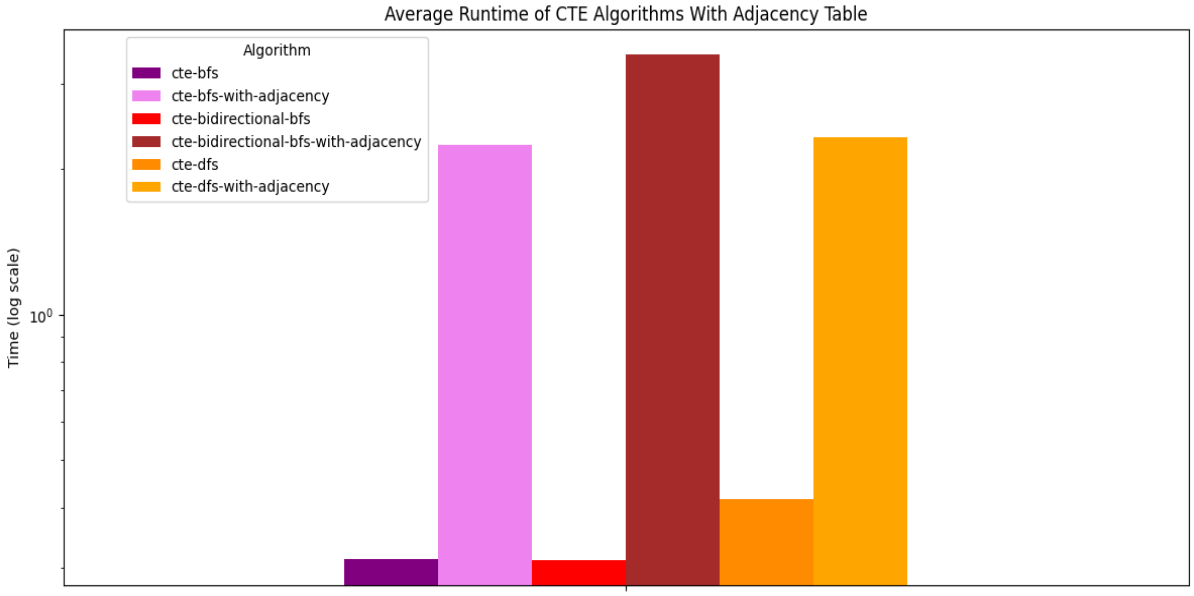

Figure 17: Average runtime of CTE in Texas Road Network

Figure 17 shows the average measurement times for each method. As observed, BFS and Bidirectional BFS have recorded nearly identical times, while the average time recorded for DFS is higher than the others. As also seen in Figure 16, the adjacency technique has led to a decline in performance.

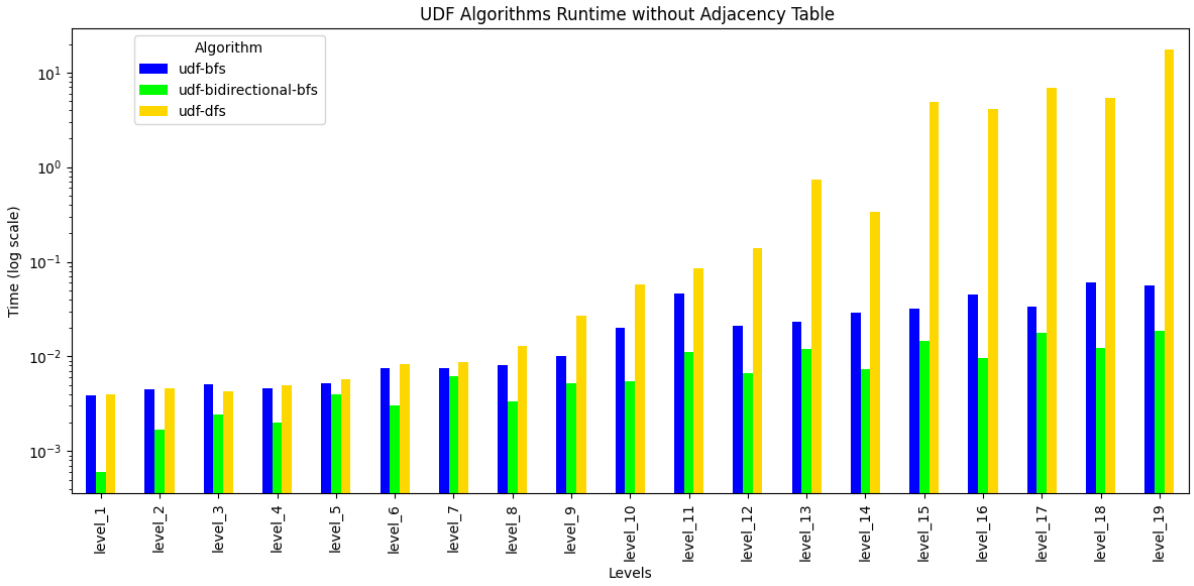

Figure 18: UDF (without adjacency) of Texas Road Network

Now, we will examine the performance of the UDF approach. As shown in Figure 18, the performance of BFS and DFS in paths with shallow depths is very similar; however, as the depth increases, BFS outperforms DFS. It is important to note that at all depths, Bidirectional BFS has consistently performed better compared to the other methods, demonstrating the best performance throughout the simulation. Although Bidirectional BFS also shows an increasing trend in measurement time as depth increases, this increase is less pronounced compared to the other methods.

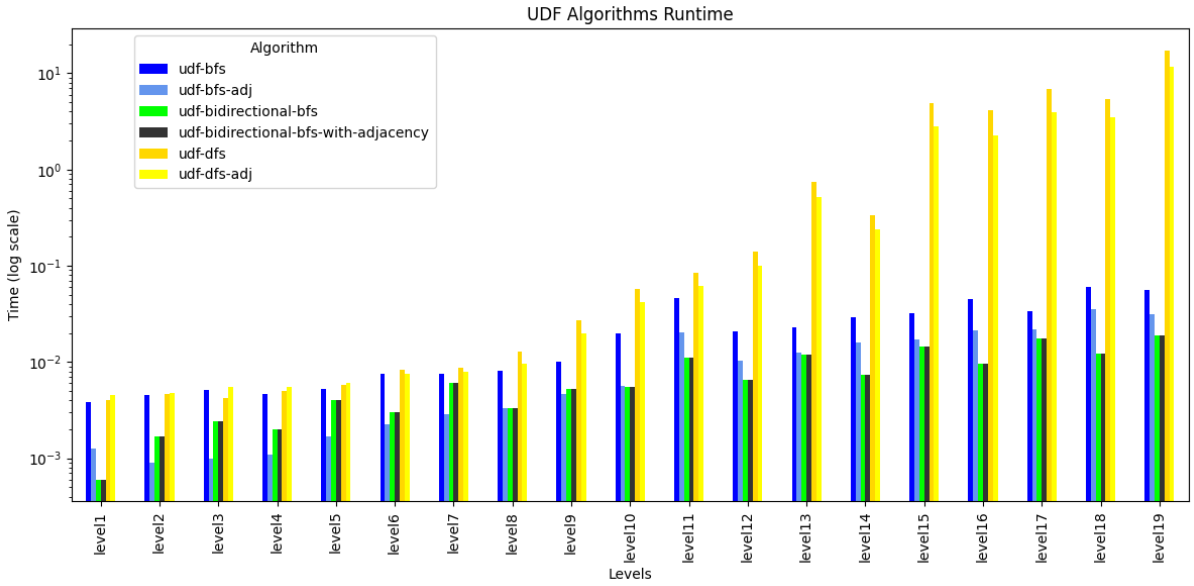

Figure 19: UDF (with adjacency) of Texas Road Network

Figure 19 analyzes the impact of the adjacency technique. As observed, the use of this technique has reduced measurement times for both BFS and DFS across all depths, leading to a significant improvement in performance. However, for Bidirectional BFS, the recorded times remain exactly the same, showing no change with the use of the adjacency technique. It is worth noting that, unlike in Figure 18 where Bidirectional BFS consistently had the best performance at all depths, here BFS with adjacency sometimes shows better performance at certain depths.

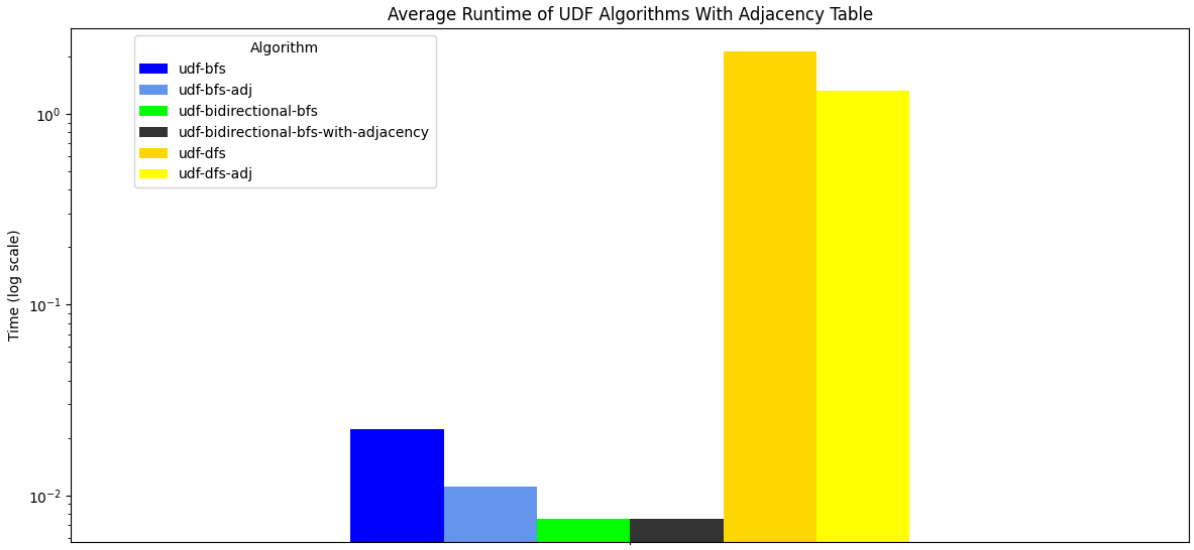

Figure 20: Average runtime of UDF in Texas Road Network

As shown in Figure 20, the average execution time for each method was measured throughout the simulations. A notable observation from this figure is the impact of the adjacency technique on both the UDF and CTE functions, which led to a reduction in measurement time. However, for the Bidirectional BFS method, no significant change was observed. As seen in Figure 19, Bidirectional BFS generally demonstrated the best performance, which is also evident here.

5 Comparing UDF and CTE

As observed in the previous section, we utilized different methods within the two main approaches, UDF and CTE, to measure the computation time for finding the shortest path. In this section, we will compare the performance of both approaches across the two datasets used in this project.

5.1 Gnutella

We will first examine the performance on the Gnutella dataset. This analysis is divided into two parts: one without using the adjacency technique and the other with using of this technique.

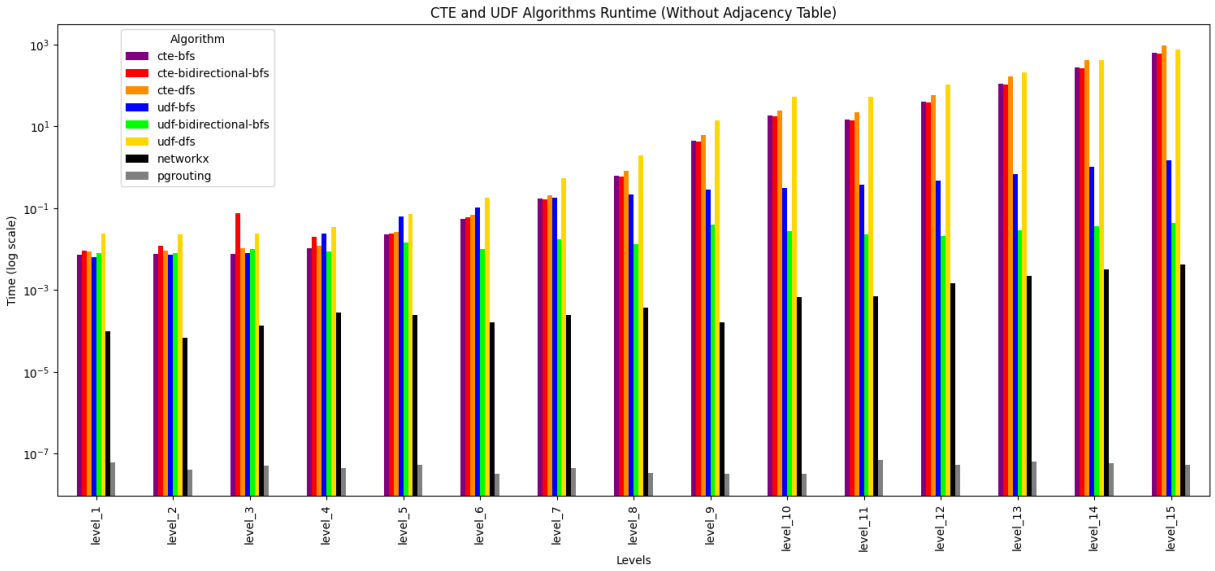

Figure 21: UDF and CTE (without adjacency) of Gneutella

As seen in Figure 21, all methods exhibit very similar performance in the first three levels, with UDF-DFS showing slightly weaker performance, though the differences between methods are not substantial. However, from level 4 onwards, the performance of all three CTE-based methods surpasses that of UDF-BFS and UDF-DFS. On the other hand, starting from level 4, UDF Bidirectional BFS demonstrates the best performance, maintaining this trend throughout the simulation.

Another notable point is the performance of CTE Bidirectional BFS, which initially lags behind the other CTE methods until level 4 but then improves, showing the best performance among CTE methods from level 7 onwards. Beyond level 8, the difference between CTE methods and UDF Bidirectional BFS and BFS becomes significant, with UDF methods demonstrating superior performance that continues to diverge as the simulation progresses. However, UDF-DFS remains weaker compared to other methods.

It's also important to note that pgRouting consistently shows the best performance, followed by NetworkX, although UDF Bidirectional BFS achieves close performance in the later levels. Both pgRouting and NetworkX demonstrate highly stable performance throughout the simulation.

Figure 22: UDF and CTE (with adjacency) of Gneutella

As shown in Figure 22, unlike Figure 21, the performance of the CTE approach in the first three levels is weaker than that of UDF, mainly due to the increased overhead introduced by the use of the adjacency technique. The performance of CTE Bidirectional BFS is particularly poor compared to other methods in this approach up to level eight, due to the additional overhead specific to this method. Beyond this level, it begins to perform closer to the other methods in this approach, although CTE BFS consistently shows the best performance among the CTE methods throughout the simulation.

Regarding the UDF methods, the performance of UDF Bidirectional BFS and UDF BFS is very close up to level 4. However, beyond this point, the execution time of UDF BFS surpasses that of UDF Bidirectional BFS and becomes comparable to the times recorded by the CTE methods. This trend continues until level 8, after which UDF BFS again demonstrates better performance compared to the CTE methods.

Throughout the simulation, UDF DFS shows the weakest performance among all methods. The comparison between these methods and the two algorithms, NetworkX and pgRouting, is also clearly depicted in the figure.

5.2 Texas Road Network

In this section, we first analyze both approaches up to level 19, similar to the previous section. Afterward, we examine their performance at higher levels.

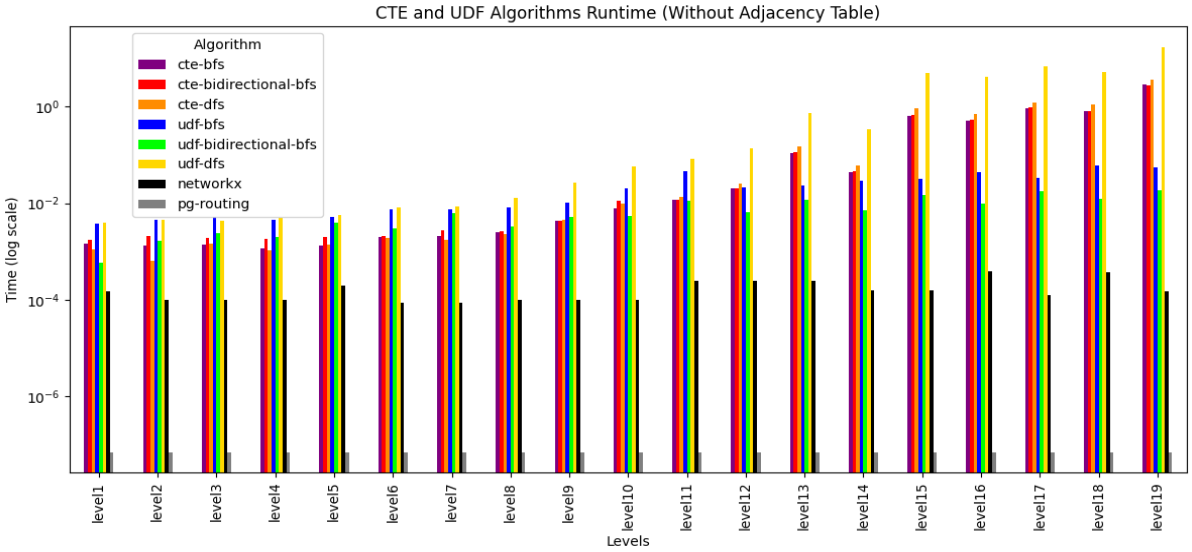

Figure 23: UDF and CTE (without adjacency) of Texas Road Network

As shown in Figure 23, all three methods of the CTE approach initially outperform BFS-UDF and DFS-UDF, and this trend continues up to level 12. However, after that point, BFS-UDF begins to perform better and maintains this advantage up to level 19. Among the three methods used in the CTE approach, DFS performs best up to level 3, but BFS outperforms from level 4 onwards, with this trend continuing until level 8. After that, bidirectional BFS, which initially had weaker performance, shows performance very close to BFS.

Regarding bidirectional BFS-UDF, it shows relatively weaker performance compared to CTE methods up to level 9, but its performance improves afterward, showing the best performance from level 9 onward. Throughout the simulation, DFS-UDF consistently shows the weakest performance.

The comparison of the methods' performance with the two algorithms, NetworkX and pgRouting, is also evident.

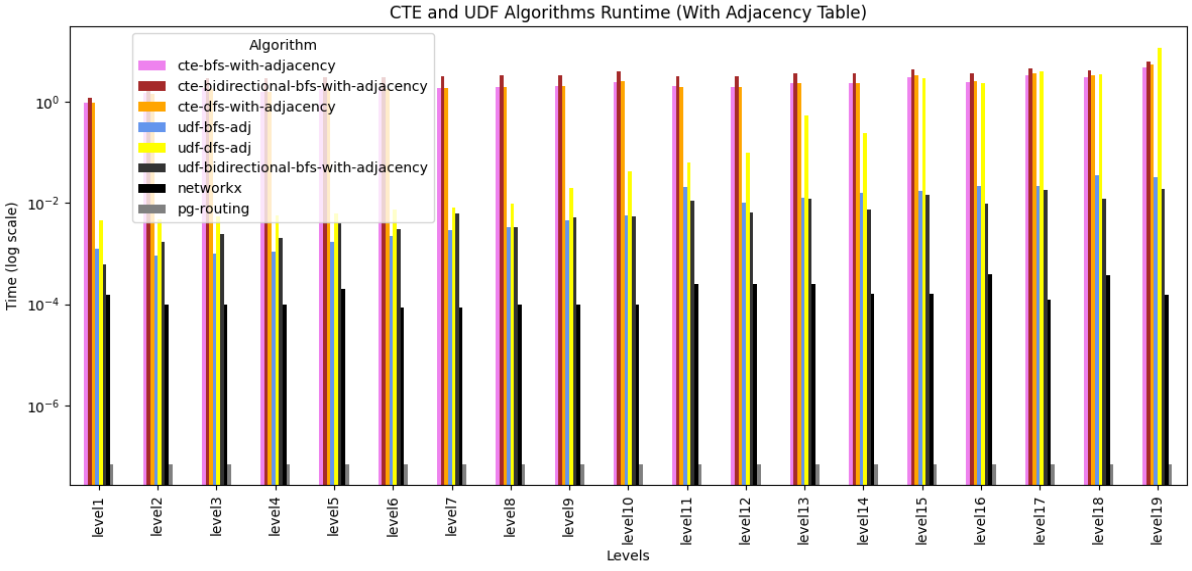

Figure 24: UDF and CTE (with adjacency) of Texas Road Network

As observed in Figure 24, the performance of CTE with the use of the adjacency technique has significantly decreased, and this decline is evident across all three methods in this approach. However, for the UDFs, the adjacency technique has improved performance in BFS and DFS. Unlike the previous chart where bidirectional BFS consistently showed the best performance among UDFs, here BFS outperforms in certain levels.

Additionally, up to level 18, all three methods of the CTE approach perform worse than the UDF approach, with only DFS-UDF showing weaker performance at level 19. The comparison with the two algorithms, pgRouting and NetworkX, is also visible in the figure.

Performance Analysis on Longer Paths

So far, we have focused on examining paths with a depth of up to 19. However, as mentioned earlier, the longest shortest path in our dataset is 1054, and there are numerous paths within our dataset that exceed a length of 19. In this section, we will analyze the performance of the CTE and UDF approaches in handling these longer paths.

Challenges with Measuring CTE Algorithms on the Texas Road Dataset

During the performance analysis of CTE algorithms on the Texas Road dataset, significant challenges arose due to the limitations of recursive queries and disk space constraints. CTEs, especially those used for recursive graph traversals, generate large intermediate results that PostgreSQL stores in temporary files on disk. Given the extensive size and complexity of the Texas Road dataset, these recursive operations quickly exhausted available disk space, leading to errors indicating that the disk was full within PostgreSQL's temporary storage directories.

Despite efforts to mitigate these issues by optimizing the recursion depth, increasing in-memory processing settings such as `work_mem` to the maximum allowable value, and freeing up disk space, the recursive nature of the queries continued to demand extensive resources beyond the capabilities of the available hardware.

Furthermore, due to these limitations, it was not possible to measure the performance of CTE algorithms beyond level 19 in the dataset. Levels beyond 19, such as level 20 and higher, could not be processed because of the overwhelming resource requirements, specifically related to disk space and the handling of recursive data structures. Consequently, the evaluation was limited to levels 0-19 to avoid the excessive disk usage that comes with deeper recursive traversals in CTEs.

In contrast, UDF approaches did not encounter the same disk space limitations in this project, as they are more efficient in managing in-memory operations without relying heavily on disk-based temporary storage. This allowed UDFs to handle the required computations more effectively within the available system resources.

Due to these persistent disk space and recursive query limitations, it was not feasible to measure the performance of CTE algorithms on levels 20 and beyond under the current setup. This underscores the need for alternative approaches or more robust hardware configurations for executing such large-scale and complex graph analyses, which are critical when dealing with datasets of this magnitude and depth.

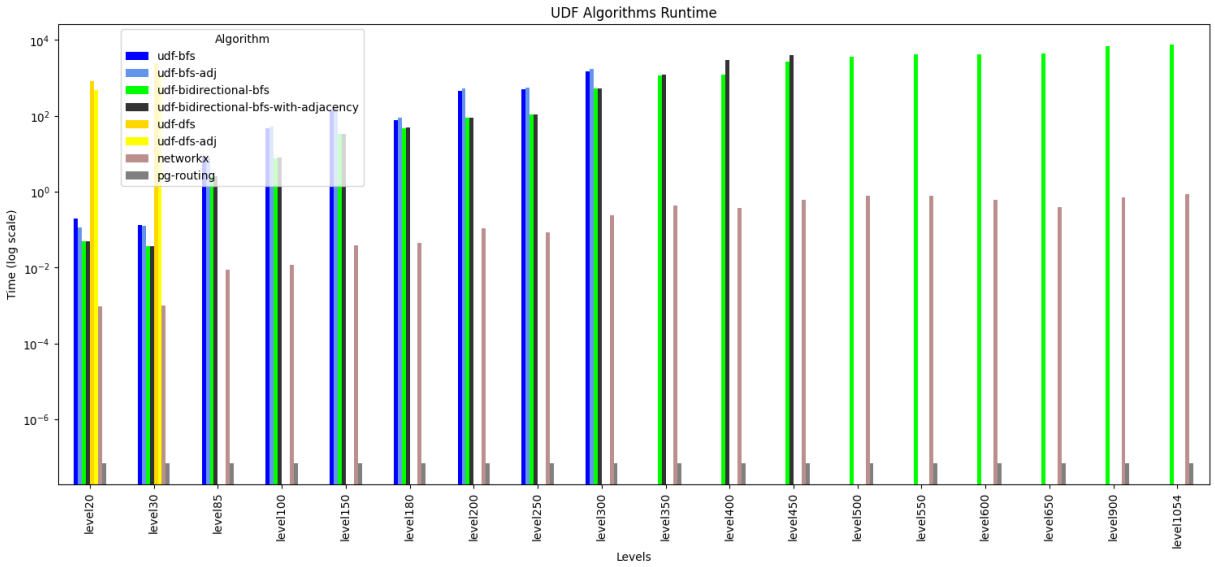

Figure 25: High level of the Texas Road

As shown in Figure 25, in this measurement, we initially examined all six UDF approach methods. However, given that the execution times for the two DFS methods were excessively high, they were only used at levels 20 and 30, and subsequently, these two methods were removed from further simulations. This approach was also applied to both BFS methods from level 300 onwards and for bidirectional BFS with the adjacency technique from level 450 onwards.

From level 450 onwards, the comparison primarily focuses on the performance of bidirectional BFS and the two algorithms, pgRouting and NetworkX. As shown in Figure 25, in the first two levels, the DFS methods using the adjacency technique demonstrate better performance. However, for bidirectional BFS, the use of this technique does not result in a noticeable difference. As mentioned earlier, from level 30 onwards, the two DFS methods were removed from the chart because, due to their nature, they exhibit weaker performance on large datasets.

For BFS at levels 20 and 30, the use of the adjacency technique improved performance; however, as the path depths increased, the performance of the two methods converged significantly. From level 100 onwards, the use of this technique began to degrade performance, and this trend continued. It is also noteworthy that, as expected, both methods showed a significant increase in execution time with increasing path depths, ultimately leading to their removal from the chart at level 300.

For bidirectional BFS, both methods initially showed identical performance, but from level 300 onwards, bidirectional BFS without the adjacency technique performed better than the method using this technique, similar to the trend observed with BFS. This trend continued until level 450, at which point the adjacency-based method was removed from the chart. It is important to note that throughout the experiments, bidirectional BFS consistently outperformed the other methods, and it was able to measure paths up to 1054, which is the longest path in this dataset. However, as the path lengths increased, there was a significant rise in the measured execution times.

Regarding the two algorithms, NetworkX and pgRouting, it is observed that they both demonstrate very stable performance throughout the simulations, particularly pgRouting, which maintains extremely low computation times for finding the shortest paths even as path lengths increase. In contrast, the execution time recorded by NetworkX slightly increases as the path lengths grow.

6 Results of simulations

As observed in both datasets, CTE performs better than UDF for paths shorter than 5, due to the lower initial overhead of the CTE approach compared to UDF. On the other hand, using the adjacency technique in CTE had the opposite effect, leading to increased path measurement times. For UDF, in the directed Gnutella dataset, performance was very close to CTE in the initial levels; however, as the path lengths increased, the UDF approach generally performed better. This trend was also seen in the Texas Road Network dataset. It is also important to note that UDF-DFS showed weaker performance compared to other methods in both datasets.

It should be noted that, unlike CTE where the adjacency technique had a negative effect, using this technique improved performance in the UDF approach for the Gnutella dataset, and similarly for the Texas Road Network, albeit only for short paths. As path lengths increased, this benefit reversed.

Another significant point, as mentioned earlier, is the limitations imposed by using the CTE method, which in this experiment restricted the maximum path length measured by the CTE approach to 19.

7 Conclusion

In this project, we explored the effectiveness of different approaches for finding the shortest paths in graph datasets, focusing on CTEs and UDFs. Our analysis highlighted the trade-offs between these methods in terms of efficiency, scalability, and adaptability to varying graph complexities.

The findings underscore the importance of selecting the right approach based on the specific needs of the dataset and computational environment. While CTEs offer a clear and structured method for short paths, their recursive nature and reliance on disk storage present significant limitations for larger or deeper graphs. UDFs demonstrated greater scalability and flexibility, particularly in handling longer paths, making them a more suitable choice for large-scale graph analysis where deeper recursive capabilities and efficient in-memory processing are essential.

Furthermore, the findings revealed that optimization methods, including the use of adjacency techniques, have to be considered in pairwise comparison with special attention to each approach and dataset. This again refers to the very important hypothesis of graph analysis: it has no general-purpose solution but is carefully balanced between algorithmic complexity, hardware capabilities, and particular characteristics of the dataset.

A significant observation throughout the study was the difference in execution times when compared to specialized graph processing libraries like pgRouting and NetworkX. The measurement times for CTEs and UDFs were substantially higher than those recorded by pgRouting and NetworkX, which maintained very low and consistent computation times even as path lengths increased.

Overall, this project contributes valuable insights into the performance dynamics of CTEs and UDFs, providing a foundation for future research and practical applications in graph database management and analysis. Further work could involve exploring hybrid models or advanced optimizations that leverage the strengths of both approaches, and incorporating specialized graph libraries like pgRouting and NetworkX to achieve even greater efficiency and performance in complex graph analyses.

Reference

[1] Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, and Alessandro Vespignani. The architecture of complex weighted networks. *Proceedings of the national academy of sciences*, 101(11):3747–3752, 2004.

[2] Shalini Batra and Charu Tyagi. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2):509–512, 2012.

[3] Itzik Ben-Gan. *T-Sql Fundamentals*. Microsoft Press, 2016.

[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[5] Author details. Advances in databases and information systems. In *Proceedings of the 2012*, 2012.

[6] Author details. Another way to implement complex computations functional-style sql udf. *Journal details*, 2012.

[7] Author details. Optimization of common table expressions in mpp database systems. *Journal details*, 2012.

[8] Author details. Using common table expressions to build a scalable boolean query generator for clinical data warehouses. *Journal details*, 2012.

[9] Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*, pages 287–290. 2022.

[10] Fifth Edition. Reinhard diestel.

[11] R Elmasri, Shamkant B Navathe, R Elmasri, and SB Navathe. Fundamentals of database systems¡/title. In *Advances in Databases and Information Systems*, volume 139. Springer, 2015.

[12] Marin Fotache and Catalin Strimbei. Sql and data analysis. some implications for data analysits and higher education. *Procedia Economics and Finance*, 20:243–251, 2015.

[13] Aric Hagberg, Pieter J Swart, and Daniel A Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

[14] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] Joseph M Hellerstein, Michael Stonebraker, James Hamilton, et al. Architecture of a database system. *Foundations and Trends® in Databases*, 1(2):141–259, 2007.

[16] Manuela Horvat and Daniel Stuparu. Common table expression - with statement. *Babes-Bolyai University, Cluj-Napoca*, 2012.

[17] Rohit Kumar Kaliyar. Graph databases: A survey. In *International Conference on Computing, Communication & Automation*, pages 785–790. IEEE, 2015.

[18] Josep Lluís Larriba-Pey, Norbert Martínez-Bazán, and David Domínguez-Sal. Introduction to graph databases. In *Reasoning Web International Summer School*, pages 171–194. Springer, 2014.

[19] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. https://snap.stanford.edu/data, 2014.

[20] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[21] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[22] D Jasmine Priskilla and K Arulanandam. An node search of dfs with spanning tree in undirected graphs. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 9(3), 2020.

[23] Md Saidur Rahman et al. *Basic graph theory*, volume 9. Springer, 2017.

[24] Karthik Ramachandra and Kwanghyun Park. Blackmagic: Automatic inlining of scalar udfs into sql queries with froid. *Proceedings of the VLDB Endowment*, 12(12):1810–1813, 2019.

[25] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. Database system concepts. 2011.

[26] Daniel Stuparu and Manuela Petrescu. Common table expression. different database systems approach. *Babes-Bolyai University, Cluj-Napoca*, 2012.

[27] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[28] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[29] Douglas B West. Introduction to graph theory, 2001.

[30] Lijing Zhang and Xuanhui He. Route search base on pgrouting. In *Software Engineering and Knowledge Engineering: Theory and Practice: Volume 2*, pages 1003–1007. Springer, 2012.