K-Shortest Path: An Analytical and Computational Approach

Muhammad Abdur Rafey (i210705)[1], Ayra Alamdar (i212968)[2], and Irtiqa Haider (i210733)[3]

[1]Computer Science, FAST National University of Computer and Emerging Sciences

April 2024

Abstract

This document presents the collaborative efforts and findings of our project focused on addressing the K-Shortest Path problem, a complex challenge in network theory that requires identifying multiple shortest paths from a single source to a single destination in a weighted graph. The significance of solving this problem lies in its extensive applications in areas such as routing, navigation, and network optimization, where multiple path options are critical for robustness and redundancy. Given the computationally intensive nature of finding multiple shortest paths, especially in large-scale networks, our project explored and implemented efficient algorithms utilizing parallel computing paradigms. We employed OpenMP for shared-memory parallelism and OpenMPI for distributed computing to enhance the algorithm's performance and scalability. This abstract outlines our approach to implementing these technologies to achieve effective parallelism and deliver faster computation times in solving the K-Shortest Path problem.

1 Introduction

The K-Shortest Path problem is a fundamental question in the field of graph theory and network analysis, involving the determination of not just the single shortest path, but the top K shortest paths from a source node to a destination node in a weighted graph. Unlike the classic shortest path problem, which seeks the single most efficient route between two points, the K-Shortest Path problem extends this inquiry to multiple paths to provide a set of the best alternative routes.

This problem has broad and significant applications across various domains. In networking, it is crucial for providing robust data transmission routes that can dynamically adapt to changes or failures in the network. In logistics, multiple efficient paths enable better risk management and cost optimization for transporting goods. Similarly, in urban planning and traffic management, identifying several optimal routes can help in evenly distributing traffic and reducing congestion, enhancing the overall efficiency of transportation systems.

Moreover, the K-Shortest Path problem is particularly relevant in scenarios where redundancy is critical for reliability and service quality. For example, in telecommunications, having multiple backup paths ensures that the network can maintain service even if one or more primary paths fail.

2 Problem Statement

The primary challenge in addressing the K-Shortest Path problem lies in its computational complexity, which increases exponentially with the size of the graph. As the number of nodes (vertices) and edges in the graph expands, the traditional algorithms used to find even a single shortest path—let alone multiple paths—become increasingly inefficient. This inefficiency is further compounded when the requirement extends to finding K shortest paths, where K can be a large number. The need for multiple paths necessitates exploring beyond the most straightforward solutions, thereby requiring more computational resources and time.

In real-world applications, where decision-making processes depend on rapid and accurate calculations—such as in real-time routing and scheduling, disaster recovery planning, or network optimization—the ability to quickly compute these paths becomes crucial. To address this complexity and meet the demands for speed and accuracy, our project leverages parallel computing techniques.

We utilize OpenMP (Open Multi-Processing) to enable multiple parallel executable threads within a single machine, harnessing the power of multi-core processors to perform computations concurrently. This approach helps in significantly reducing the computation time by dividing the task among several processors working in parallel, each handling a portion of the data or operations.

In addition to OpenMP, we employ OpenMPI (Message Passing Interface), which allows for the problem to be broken down into smaller, manageable parts. This technique is particularly useful when the task at hand exceeds the capacity of a single machine or when it can be efficiently solved using a divide-and-conquer strategy. Using OpenMPI, a master process assigns tasks to slave processes, which can be located on the same machine or distributed across a cluster of machines. Each slave process solves a part of the problem independently, and the results are then aggregated by the master process to produce the final outcome.

By integrating OpenMP and OpenMPI, we aim to tackle the scalability issues inherent in the K-Shortest Path problem, enabling our solution to handle larger graphs and higher values of K effectively. The combination of these two parallelization strategies ensures that our approach not only scales with the problem size but also delivers faster and more reliable results, which are essential for the practical application of the K-Shortest Path analysis in dynamic and complex network environments.

3 Overview

In our investigation of the K-Shortest Path problem, our team reviewed several existing algorithms and techniques known for their efficacy in solving pathfinding problems in graphs. The selection process involved a comparative analysis of algorithms based on their complexity, scalability, and suitability for parallelization. Among the considered algorithms were the Yen's algorithm, the Eppstein algorithm, and the Bellman-Ford algorithm, each having distinct characteristics and performance profiles in different scenarios.

4 Possible Solutions

The algorithms evaluated for solving the K-Shortest Path problem include the following:

- **Yen's Algorithm**: Notable for its ability to find the K-shortest loopless paths in a graph. It builds upon Dijkstra's algorithm, iteratively discovering new paths based on the shortest path found in the previous iteration. This characteristic makes Yen's algorithm particularly amenable to enhancements in computational efficiency through parallel processing.

- **Eppstein's Algorithm**: Excels in scenarios where the graph does not change frequently, and multiple queries regarding the shortest paths are required. It uses a more complex data structure to store potential paths, making the retrieval of each successive path computationally cheaper than recalculating everything from scratch.

- **Bellman-Ford Algorithm**: Typically employed for graphs that include edges with negative weights, offering a robust solution that also detects negative cycles in the graph. Though not as efficient as Dijkstra's for non-negative graph scenarios, it provides a comprehensive solution where other algorithms might fail.

- **A* Algorithm**: A popular heuristic-based approach commonly used in pathfinding and graph traversals. Unlike typical shortest path algorithms, A* uses heuristics to estimate the cost of reaching the destination from each node, significantly speeding up the search process by exploring only the most promising paths. This makes it particularly effective in large-scale applications where paths need to be computed dynamically, such as in real-time systems and games. However, A* requires a well-designed heuristic that is appropriate to the specificities of the graph to perform optimally, and its efficiency may decrease as the complexity of the heuristic increases.

Each of these algorithms has its advantages and potential drawbacks depending on the specific characteristics of the problem being addressed. After analyzing their theoretical foundations and practical implications, our team decided to proceed with Yen's Algorithm due to its balance between computational efficiency and ease of parallelization, which was critical for our project objectives.

5 Used Solution

After extensive evaluation of various algorithms for the K-Shortest Path problem, our group decided to implement Yen's Algorithm. This decision was based on a series of practical constraints and performance considerations associated with the alternatives we explored.

- **A* Algorithm**: Initially considered due to its efficient heuristic-based approach, A* was eventually ruled out because it required the use of a large 2D matrix for heuristic evaluations. The matrix size needed to exceed 10,000 x 10,000 elements to handle our graph's complexity, which was impractical for storage in RAM. Additionally, managing such a large matrix and distributing it across a cluster via CSV files proved excessively challenging, impacting the feasibility of implementation.

- **Bellman-Ford Algorithm**: While known for its ability to handle graphs with negative weights, the Bellman-Ford algorithm was less suitable for our needs due to its higher computational time. Given that our input data did not include negative weights, the additional complexity and runtime of Bellman-Ford were unnecessary.

- **Eppstein's Algorithm**: Similar to A*, Eppstein's algorithm also required the creation of a large matrix to store potential paths. The significant memory requirement and the complexity of managing such data structures were major deterrents, given our computational resources and the nature of the problem we aimed to solve.

Yen's Algorithm, in contrast, utilizes an adjacency list approach, which is far more memory-efficient for our data size and complexity. This algorithm builds upon the well-established Dijkstra's method to iteratively find multiple shortest paths without the excessive overhead of handling large matrix structures. Its compatibility with parallel processing techniques also made Yen's an ideal choice. We were able to implement it effectively using both serial and parallel approaches, significantly enhancing performance while managing resource constraints.

6 Algorithm Explanation

The algorithm used in our project to find the K-shortest paths is a modification of Yen's algorithm that leverages Dijkstra's algorithm for the shortest path computations. Here is the pseudocode detailing the approach:

```
function findKShortestPaths(start, end, k):
    vector<vector<pair<int, int>>> shortestPaths
    vector<pair<int, int>> shortestPath = Dijkstra(start, end)
    shortestPaths.push_back(shortestPath)

    k = k - 1
    while k > 0:
        Graph g = copy of current Graph

        vector<pair<int, int>> excludedEdges(size of shortestPaths)

        for i from 0 to size of shortestPaths:
            randomEdge = random index in shortestPaths[i]
            excludedEdges[i] = (shortestPaths[i][randomEdge],
                                shortestPaths[i][randomEdge + 1])

        vector<pair<int, int>> path = g.Dijkstra(start, end, excludedEdges)

        if path not in shortestPaths:
            if size of path > 1:
                shortestPaths.push_back(path)
                k = k - 1

    return shortestPaths

function Dijkstra(start, end, excludedEdges = empty vector):
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>
    vector<int> distance(Size, INT_MAX)
    vector<int> parent(Size, -1)

    distance[start] = 0
    pq.push((0, start))
```

```
while not pq.empty():
    u = pq.top().second
    pq.pop()

    for each (v, weight) in graph[u]:
        if (u, v) not in excludedEdges:
            if distance[u] + weight < distance[v]:
                distance[v] = distance[u] + weight
                parent[v] = u
                pq.push((distance[v], v))

vector<pair<int, int>> path
for v = end; v != -1; v = parent[v]:
    path.push_back((v, distance[v]))

reverse path
return path
```

This pseudocode outlines the primary functions used to implement Yen's algorithm for finding multiple shortest paths. It utilizes an adapted version of Dijkstra's algorithm for pathfinding, with modifications to exclude specific edges and thereby generate alternative paths. This approach effectively produces the K-shortest paths between a source and destination in a graph.

7 OpenMPI and OpenMP Utilization

In our project, we utilized OpenMPI and OpenMP to optimize the performance and efficiency of the K-Shortest Path algorithm. Below are the specific ways in which each technology was applied:

- **OpenMP Usage (Open Multi-Processing):** OpenMP was primarily used to enable shared-memory parallelism within a single compute node. Specific implementations include:
  - *Parallel Edge Exclusion:* Each thread randomly selects an edge from the previously identified shortest paths to exclude in the next iteration. This process is parallelized to ensure simultaneous edge exclusion across different paths, significantly reducing iteration time.
  - *Concurrent Dijkstra Execution:* Dijkstra's algorithm is executed concurrently across multiple threads, with each thread processing a separate portion of the graph. This utilization of multiple cores speeds up the pathfinding computation.
- **OpenMPI Usage (Open Message Passing Interface):** OpenMPI facilitated distributed memory parallelism across multiple compute nodes, effectively reducing the computational load on any single machine. Key implementations include:
  - *Data Distribution:* The master node reads the graph data once and distributes it across all slave nodes. This minimizes I/O overhead and ensures consistent data for computation across all nodes.
  - *Task Distribution:* The master node distributes the start and end vertices for the K-Shortest Path computations to each slave node. This distribution allows simultaneous handling of different parts of the problem, speeding up the overall computational process.

Combined Benefits: The integration of OpenMPI and OpenMP enabled effective scaling of the K-Shortest Path algorithm through both intra-node (via OpenMP) and inter-node (via OpenMPI) parallelism. This multi-level approach utilized maximum computational resources, decreased execution times, and effectively managed large graph datasets. Additionally, by distributing tasks across a computer cluster, it addressed memory constraints that typically affect single-node computations.

8 Execution Time Comparison

The following table and figure illustrate the average parallel execution times for different numbers of threads used in processing.

Thread Number	Average Parallel Execution Time
2	1854.8ms
3	1529.4ms
4	1098.8ms
5	1596.4ms
6	2425.4ms
7	1827.8ms
8	455ms
9	940ms

Table 1: Comparison of execution times for different numbers of threads.

Graph Size	Average Serial Execution Time	Average Parallel Execution Time
4	0ms	1280ms
5	0ms	1198.4ms
6	0ms	31.2ms
7	0ms	434.8ms
10	0ms	601.4ms
20	0ms	305.4ms
36692	33.4ms	46.8ms
265214	39ms	40ms

Table 2: Comparison of execution times for Serial and Parallel.

Parallel% Speedup	No of Processers
0.1467	3
1.108	
0.1467	5
1.132	
0.1467	7
1.143	
0.1467	9
1.149	

Table 3: Speedup Calculation by using Amdahl's law

9 Speed Up Calculation

Calculation using Amdahl's Law:

$$\text{Speedup} = \frac{1}{(1-f)+\frac{f}{n}}$$

Calculation using Gustafson's Law:

Speedup = N + (1-N) x s

Calculation using Time Observed:

As we already have serial and parallel execution time available, we can calculate the speedup simply bu using formula:

$$\text{Speedup} = \frac{SerialExecutionTime}{ParallelExecutionTime}$$

Serial % Speedup	No of Processers
0.8533	3
1.2934	
0.8533	5
1.5868	
0.8533	7
1.8802	
0.8533	9
2.1736	

Table 4: Speedup Calculation by using Gustafson's law

Graph Size	Speedup
4	0
5	0
6	0
7	0
10	0
20	0
36692	0.71
265214	0.97

Speedup Calculation by using $\frac{Ts}{Tp}$

10 Conclusion

Our project's exploration into the K-Shortest Path problem using Yen's Algorithm provided substantial insights into the complexities and computational demands of determining multiple shortest paths in large-scale graphs. The challenges faced were multifaceted, ranging from data preprocessing to the technical intricacies of parallel computation.

Data Preprocessing Challenges: One of the initial hurdles was cleaning the graph data to ensure accurate computations. This involved removing self-loops and duplicate edges from the adjacency list, which are critical steps to prevent skewing the results and to optimize processing efficiency. The cleanliness of data directly impacted the performance of the algorithm, underscoring the importance of robust data preprocessing protocols in graph-related algorithms.

Parallel Computation Challenges: The utilization of OpenMPI highlighted limitations in the MPI send and receive functions, particularly the limited buffer size for the arrays that could be sent across processes. This limitation required us to innovate around data partitioning and communication strategies, ensuring efficient data handling without overwhelming the buffer capacities.

Algorithmic Challenges: Testing different algorithms to find a suitable heuristic for the A* algorithm was another significant challenge. Despite the theoretical promise of A*, finding an effective heuristic that could be generalized across diverse datasets proved unfruitful in our context. This experience highlighted the need for a more adaptable heuristic evaluation strategy that could dynamically adjust based on the graph's characteristics.

Future Work: The findings from this project suggest several avenues for future research: 1. *Advanced Data Cleaning Techniques:* Developing more sophisticated data preprocessing tools that can automatically detect and correct anomalies in graph data could further streamline the computational process. 2. *Buffer Management in MPI:* Researching and possibly developing enhancements to MPI's buffer management could alleviate the limitations faced during the project, possibly improving the scalability and efficiency of parallel computing in graph algorithms. 3. *Heuristic Development for A*:* Further research into adaptive heuristic functions for A* could yield a more robust and versatile approach, potentially making A* more applicable to a broader range of problems in graph theory.

In conclusion, while the project faced several challenges, it also opened up new pathways for improvement and innovation in the field of graph algorithms and parallel computing. The experience has laid a solid foundation for further exploration and development in handling complex graph-based problems efficiently.

11 Requirements

Before compiling and running the program, ensure that the following requirements are met:

- MPI implementation (e.g., MPICH or OpenMPI) is installed.
- GNU Compiler Collection (GCC) with OpenMP support is installed.

12 Compilation

Compile the program using the following command:

```
mpic++ -fopenmp K-Shortest.cpp -o run
```

Explanation of the command:

- `mpic++`: The MPI C++ compiler.
- `-fopenmp`: Compiler flag to enable OpenMP.
- `K-Shortest.cpp`: The source file.
- `-o run`: Specifies the output executable file name.

13 Execution

Execute the compiled program with the following command:

```
mpiexec -n <number of workers> ./run <File Name> <File Type> <Value of K>
```

Replace the placeholders with appropriate values:

- `<number of workers>`: Number of processes to use.

- `<File Name>`: Path to the input file.
- `<File Type>`: Type of the input file.
- `<Value of K>`: The number of shortest paths to compute.

14 References

- Yen's Algorithm - Wikipedia
- Article on Wiley Online Library
- David Eppstein's Publication on Finding the k Shortest Paths
- GeeksforGeeks: Shortest Path by Removing K Walls

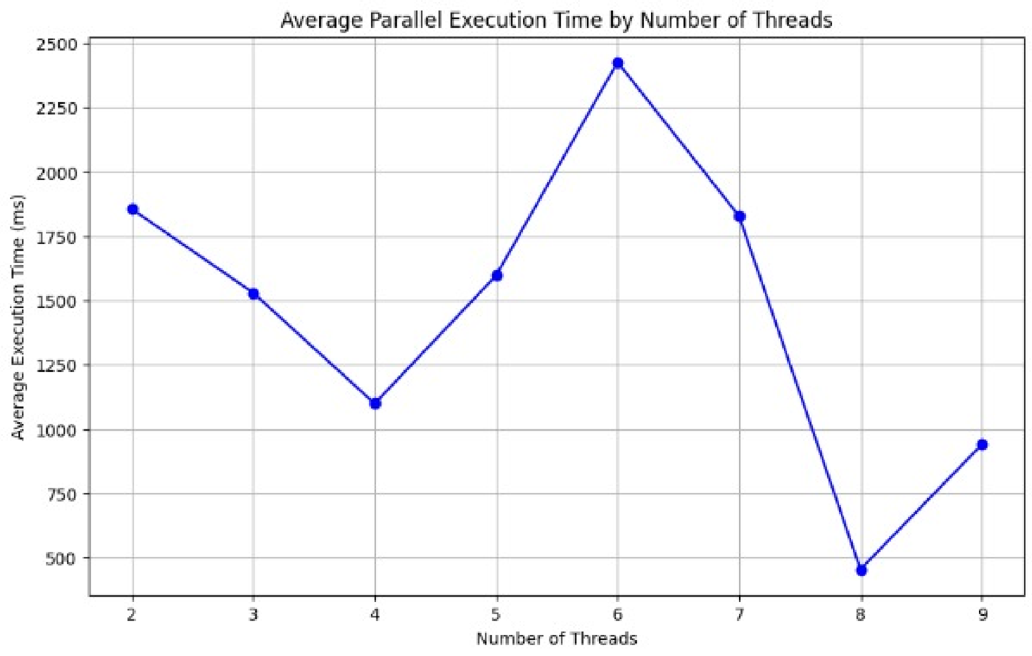

Figure 1: Thread Execution Over Time

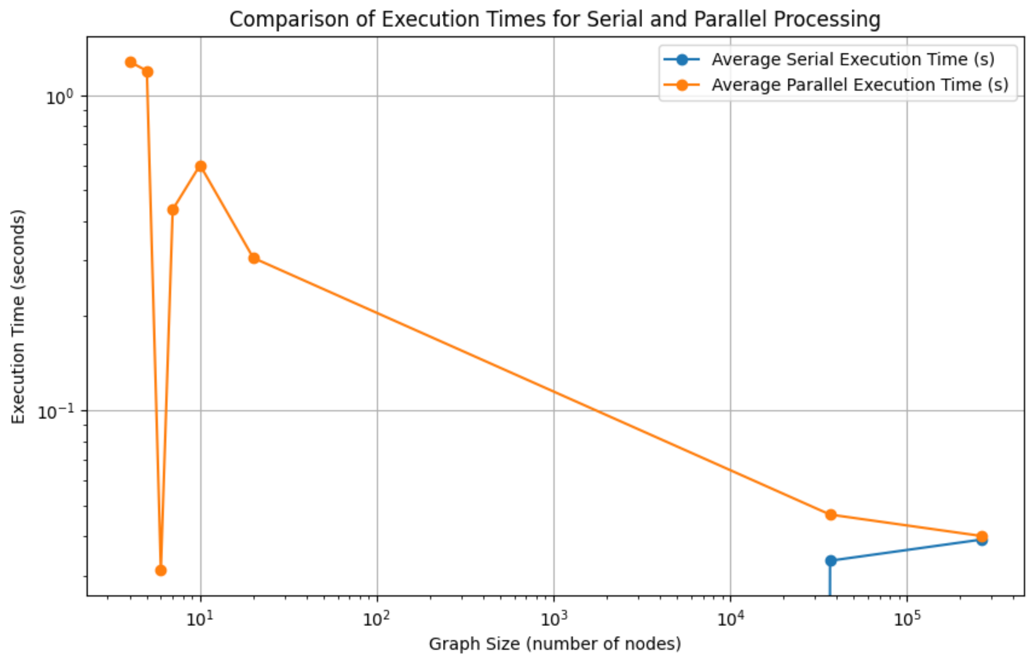

Figure 2: Thread Execution Over Time