Project Report

2023

Design and Analysis of Algorithms

Made By:

Minam Faisal (21i-1901)

Momenah Saif (21i-1909)

Submission Date: 5th December, 2023

Codes Explanation:-

1. *Normalization:*

- Data Structure Definition:

  Defines a structure named Data that has four members: column1, column2, column3, and column4.

- File Reading Function:

  Declares a function called Filereading that reads data from a file specified by the filename parameter and stores it in a vector of Data structures. Opens the file using an input file stream and checks if it's successfully opened. Reads each line from the file, extracts four values using sscanf, creates a Data structure, and adds it to the vector.

- Printing Function:

  Defines a function called print that prints the contents of a vector of Data structures.

- Execution Time Measurement Function:

  Declares a function named ExecutionTimeCalculation that measures the execution time of a given function using the <chrono> library. Records the start time, executes the provided function, records the end time, and calculates the duration.

- Main Function:

  Declares a vector named dataVector to store the data read from the file. Measures the execution time of the file reading function using the ExecutionTimeCalculation function and prints the time taken. Prints the contents of the vector using the print function.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <functional>
using namespace std;

//This representing different columns of data
struct Data
 {
    int col1;
    int col2;
    int col3;
    long long col4;
};


//This function reads a file and processes its contents it takes a filename and a reference to a vector of Data as parameters.
void Filereading(const string& filename, vector<Data>& dataVector)
{
      //Attempts to open the file using an input filestream
      ifstream file(filename);
    if (!file.is_open())
     {
        cout << "Error opening the file: " << filename << endl;
       return;
    }
      //Reads each line from the file, extracts four values using sscanf, creates a Data structure, and adds it to the vector.
   string line;
    while (getline(file, line))
     {
        Data data;
        sscanf(line.c_str(), "%d\t%d\t%d\t%lld", &data.col1, &data.col2, &data.col3, &data.col4);
        dataVector.push_back(data);
    }

    file.close();
}

//This function measures the execution time of a given function using the <chrono> library
void print(const vector<Data>& dataVector)
{
    for (const auto& data : dataVector)
    {
        cout << data.col1 << "\t" << data.col2 << "\t" << data.col3 << "\t" << data.col4 << endl;
    }
}

double ExecutionTimeCalculation(function<void()> function)
{
    auto start =chrono::high_resolution_clock::now();
    function();
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    return duration.count();
}

int main()
{
    //vector of Data named dataVector is declared.
    vector<Data> dataVector;

```

Output:

- Time taken to read and process data in csv file = 0.00654087

```
┌──(kali㉿kali)-[~/Desktop/algo_project]
└─$ g++ Normalization.cpp -o n

┌──(kali㉿kali)-[~/Desktop/algo_project]
└─$ ./n
Time taken to read and process data: 0.00654087 seconds
1	0	0	0
2	0	0	0
3	0	0	0
0	0	0	0
3	0	0	0
2	0	0	0
4	0	0	0
4	0	0	0
3	0	0	0
4	0	0	0
0	0	0	0
0	0	0	0
3	0	0	0
4	0	0	0
1	0	0	0
0	0	0	0
1	0	0	0
5	0	0	0
4	0	0	0
2	0	0	0
4	0	0	0
0	0	0	0
8	0	0	0
6	0	0	0
1	0	0	0
3	0	0	0
2	0	0	0
1	0	0	0
7	0	0	0
0	0	0	0
4	0	0	0
0	0	0	0
2	0	0	0
1	0	0	0
```

2. *Graph Representation:*

- ### File Reading:

  The Filereading function reads data from a CSV file and stores it in a vector of structures Data. It checks if the file can be opened and reads each line of the file. If any line cannot be read correctly, it catches the error and continues processing the rest of the file.

- ### Data Structure:

  The program uses a structure called Data to represent each record in the file. Each record has information about the source, destination, weight, and timestamp of a connection.

- ### Sorting:

  The MergeSorting function uses a stable merge sort algorithm to sort the data based on the timestamp. This ensures that records with equal timestamps maintain their original order.

- ### File Writing:

  The Filewriting function writes the sorted data back to a file in CSV format.

- ### Graph Representation:

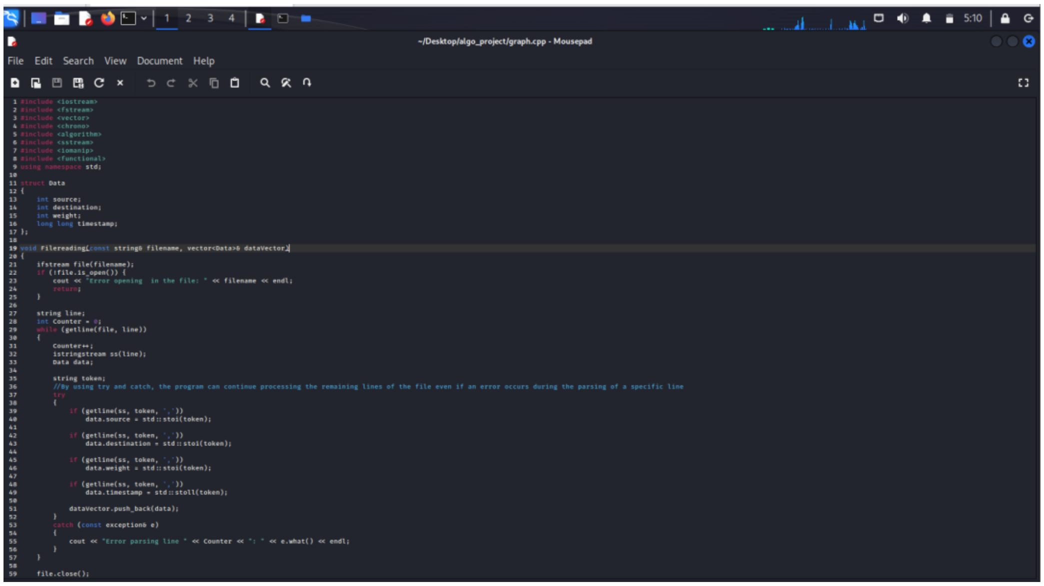
  The displayGraph function outputs a representation of the graph connections to the console. It shows the source and destination of each connection.

- ### Timing Measurement:

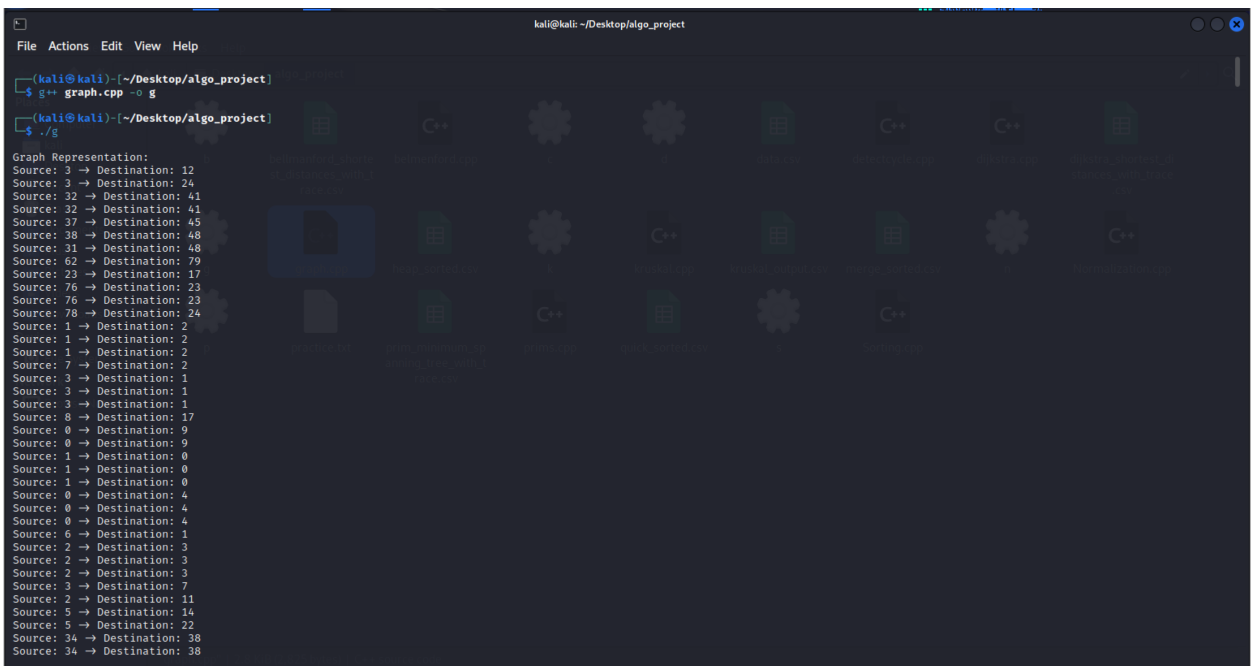
  The Timemeasurement function measures the time it takes to execute a given function.

- ### Main Function:

  In the main function, reads data from "data.csv" into the dataVector. Sorts the data based on timestamp using merge sort. Displays the graph representation on the terminal.



Output:



3. *Sorting:*

- Data Structure:

  The Data struct represents a connection for source, destination, weight, and timestamp.

- File Reading:

  The Filereading function reads data from a CSV file into a vector of Data structures. It handles errors if the file cannot be opened or if there's an issue in reading the data.

- Sorting Algorithms:

  Three sorting algorithms:-

  Quick Sort: It's a fast sorting algorithm that uses a divide-and-conquer strategy.
  Merge Sort: It's a stable sorting algorithm that also uses a divide-and-conquer approach.
  Heap Sort: It's a comparison-based sorting algorithm that uses a binary heap data structure.

- Sorting Functions:

  QuickSorting, MergeSorting, and HeapSorting functions sort the data vector using their respective sorting algorithms.

- File Writing:

  The Filewriting function writes the sorted data to a new CSV file.

- Display Function:

  The displayGraph function prints the data in a tabular format on the terminal.

- Time Measurement:

  The Timemeasurement function measures the execution time of a provided function using the C++ <chrono> library.

- Main Function:

  Reads data from "data.csv" into dataVector. Applies Quick Sort, measures the time taken, and displays, saves the sorted data. Applies Merge Sort, measures the time taken, and displays, saves the sorted data. Applies Heap Sort, measures the time taken, and display, saves the sorted data. Each step involves printing the execution time and displaying the sorted data.

- Time Complexities of Sorting Algorithms Analysis:

- ***Merge Sort:***

  Best Case Time Complexity: O (n log n)

  Worst Case Time Complexity: O (n log n)

  Average Case Time Complexity: O (n log n)

Merge sort has a consistent time complexity of O (n log n) for all cases because it always divides the array into two halves and recursively sorts them before merging.

- ***Heap Sort:***

  Best Case Time Complexity: O (n log n)

  Worst Case Time Complexity: O (n log n)

  Average Case Time Complexity: O (n log n)

Heap sort also has a consistent time complexity of O (n log n) for all cases. The build-heap operation has a time complexity of O (n), and the heapify operation in the sorting phase has a time complexity of O (log n).

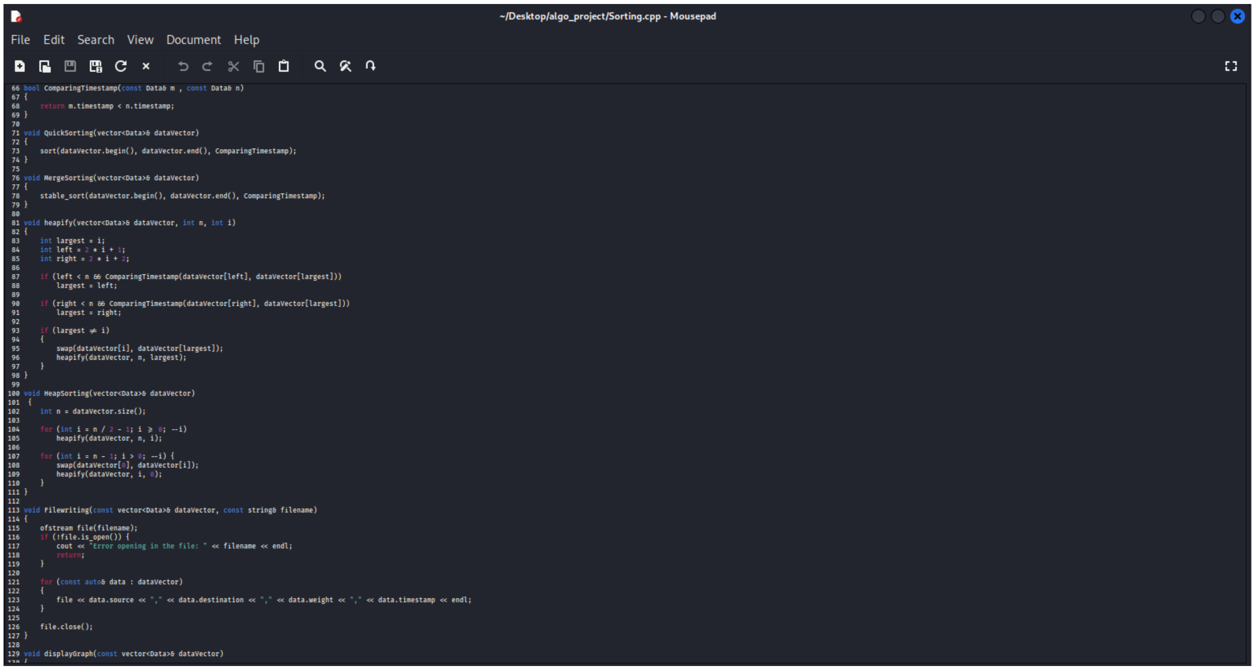
- ***Quick Sort:***

  Best Case Time Complexity: O (n log n)

  Worst Case Time Complexity: O (n^2)

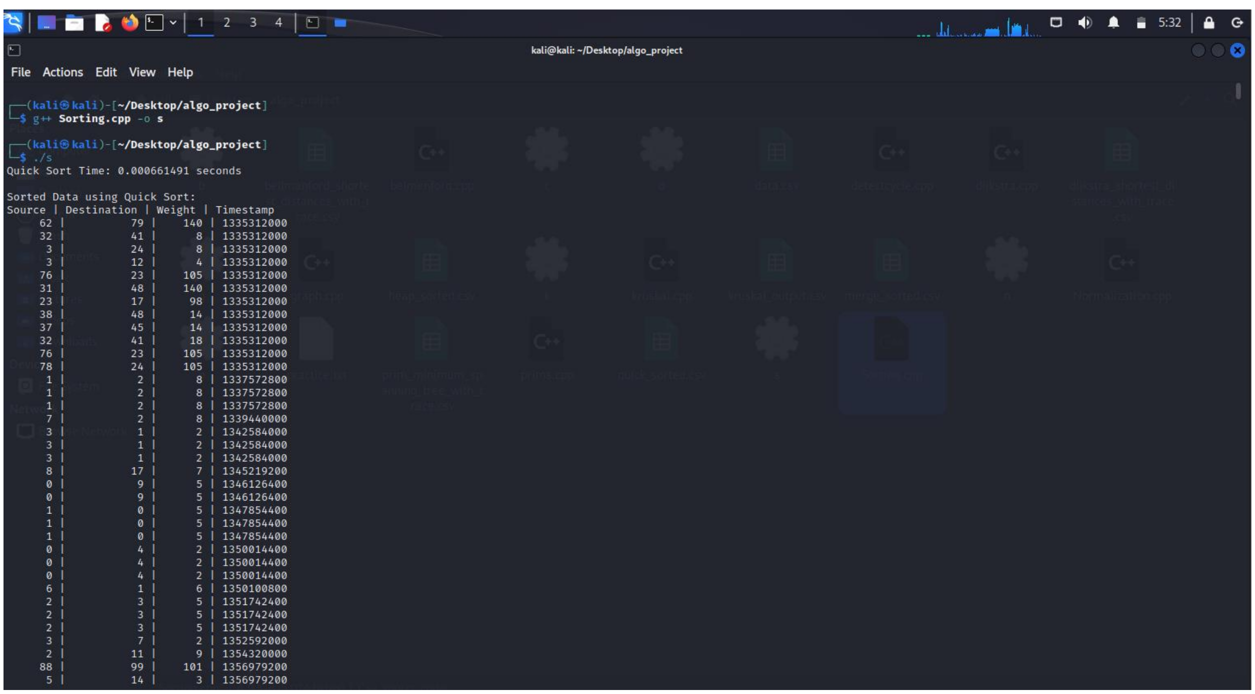
  Average Case Time Complexity: O (n log n)

Quick sort has a best-case time complexity of O (n log n) when the pivot is always chosen as the median element. However, in the worst case, if the pivot is consistently chosen poorly (e.g., always the smallest or largest element), the time complexity degrades to O (n^2). On average, with a good pivot selection strategy, the time complexity is O (n log n).

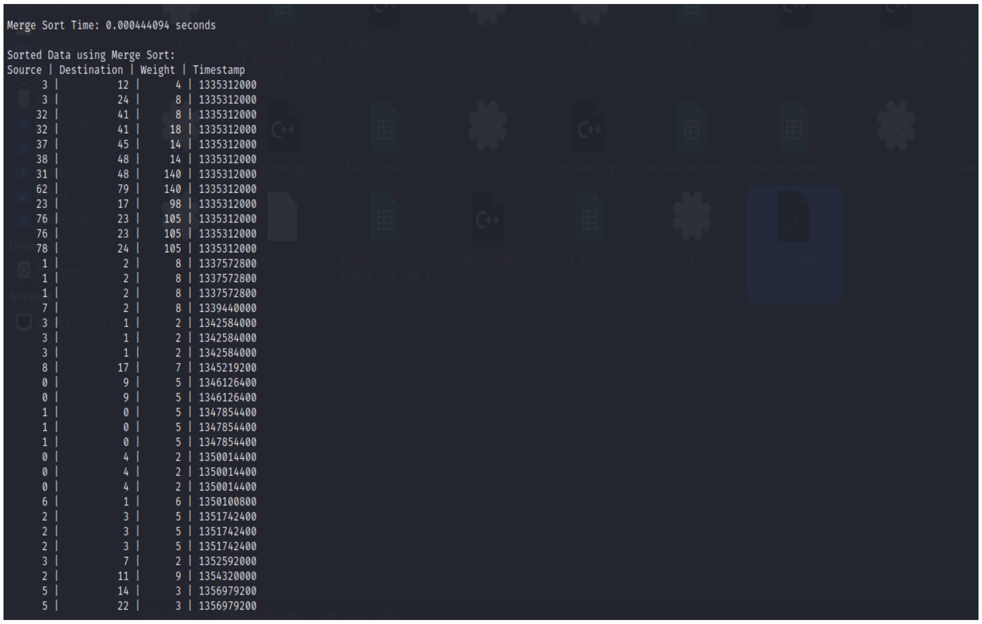


Output:

- Quick Sort Time = 0.000661491

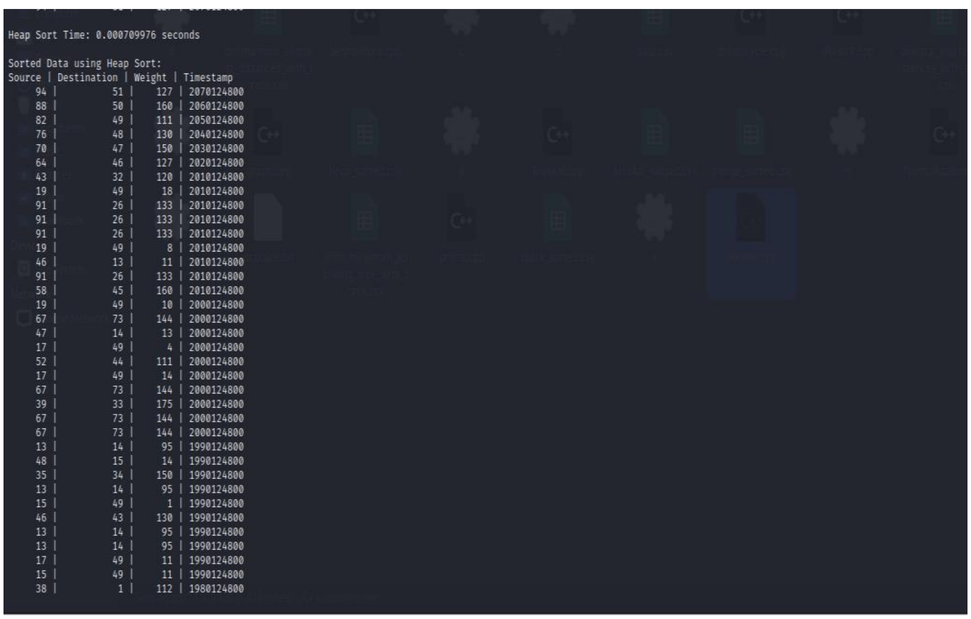


- Merge Sort Time = 0.000444094



```
Merge Sort Time: 0.000444094 seconds

Sorted Data using Merge Sort:
Source | Destination | Weight | Timestamp
     3 |          12 |      4 | 1335312000
     3 |          24 |      8 | 1335312000
    32 |          41 |      8 | 1335312000
    32 |          41 |     18 | 1335312000
    37 |          45 |     14 | 1335312000
    38 |          48 |     14 | 1335312000
    31 |          48 |    140 | 1335312000
    62 |          79 |    140 | 1335312000
    23 |          17 |     98 | 1335312000
    76 |          23 |    105 | 1335312000
    76 |          23 |    105 | 1335312000
    78 |          24 |    105 | 1335312000
     1 |           2 |      8 | 1337572800
     1 |           2 |      8 | 1337572800
     1 |           2 |      8 | 1337572800
     7 |           2 |      8 | 1339440000
     3 |           1 |      2 | 1342584000
     3 |           1 |      2 | 1342584000
     3 |           1 |      2 | 1342584000
     8 |          17 |      7 | 1345219200
     0 |           9 |      5 | 1346126400
     0 |           9 |      5 | 1346126400
     1 |           0 |      5 | 1347854400
     1 |           0 |      5 | 1347854400
     1 |           0 |      5 | 1347854400
     0 |           4 |      2 | 1350014400
     0 |           4 |      2 | 1350014400
     0 |           4 |      2 | 1350014400
     6 |           1 |      6 | 1350100800
     2 |           3 |      5 | 1351742400
     2 |           3 |      5 | 1351742400
     2 |           3 |      5 | 1351742400
     3 |           7 |      2 | 1352592000
     2 |          11 |      9 | 1354320000
     5 |          14 |      3 | 1356979200
     5 |          22 |      3 | 1356979200
```

- Heap Sort Time = 0.000709976



```
Heap Sort Time: 0.000709976 seconds

Sorted Data using Heap Sort:
Source | Destination | Weight | Timestamp
    94 |          51 |    127 | 2070124800
    88 |          50 |    160 | 2060124800
    82 |          49 |    111 | 2050124800
    76 |          48 |    130 | 2040124800
    70 |          47 |    150 | 2030124800
    64 |          46 |    127 | 2020124800
    43 |          32 |    120 | 2010124800
    19 |          49 |     18 | 2010124800
    91 |          26 |    133 | 2010124800
    91 |          26 |    133 | 2010124800
    91 |          26 |    133 | 2010124800
    19 |          49 |      8 | 2010124800
    46 |          13 |     11 | 2010124800
    91 |          26 |    133 | 2010124800
    58 |          45 |    160 | 2010124800
    19 |          49 |     10 | 2000124800
    67 |          73 |    144 | 2000124800
    47 |          14 |     13 | 2000124800
    17 |          49 |      4 | 2000124800
    52 |          44 |    111 | 2000124800
    17 |          49 |     14 | 2000124800
    67 |          73 |    144 | 2000124800
    39 |          33 |    175 | 2000124800
    67 |          73 |    144 | 2000124800
    67 |          73 |    144 | 2000124800
    13 |          14 |     95 | 1990124800
    48 |          15 |     14 | 1990124800
    35 |          34 |    150 | 1990124800
    13 |          14 |     95 | 1990124800
    15 |          49 |      1 | 1990124800
    46 |          43 |    130 | 1990124800
    13 |          14 |     95 | 1990124800
    13 |          14 |     95 | 1990124800
    17 |          49 |     11 | 1990124800
    15 |          49 |     11 | 1990124800
    38 |           1 |    112 | 1980124800
```

4. *Dijkstra:*

- Introduction:

The code is written to implement Dijkstra's algorithm, a method for finding the shortest paths between nodes in a directed graph. The graph is defined by a set of nodes and edges, each with an associated weight.

- Data Reading:

The program starts by reading information about nodes and edges from a CSV file called "data.csv". Each line in the file represents a connection between two nodes, with a numerical weight. The code ensures that invalid or out-of-range values are handled properly.

- Graph Representation:

The graph is represented using an adjacency list, a data structure that stores each node's neighbors and their corresponding weights. This representation allows for efficient traversal of the graph.

- Dijkstra's Algorithm:

The main part of the code is the implementation of Dijkstra's algorithm. This algorithm finds the shortest paths from a specified source node to all other nodes in the graph.

- Time Measurement:

The code includes a function to measure the execution time of Dijkstra's algorithm. This helps evaluate the efficiency of the algorithm.

- Results Display and Storage:

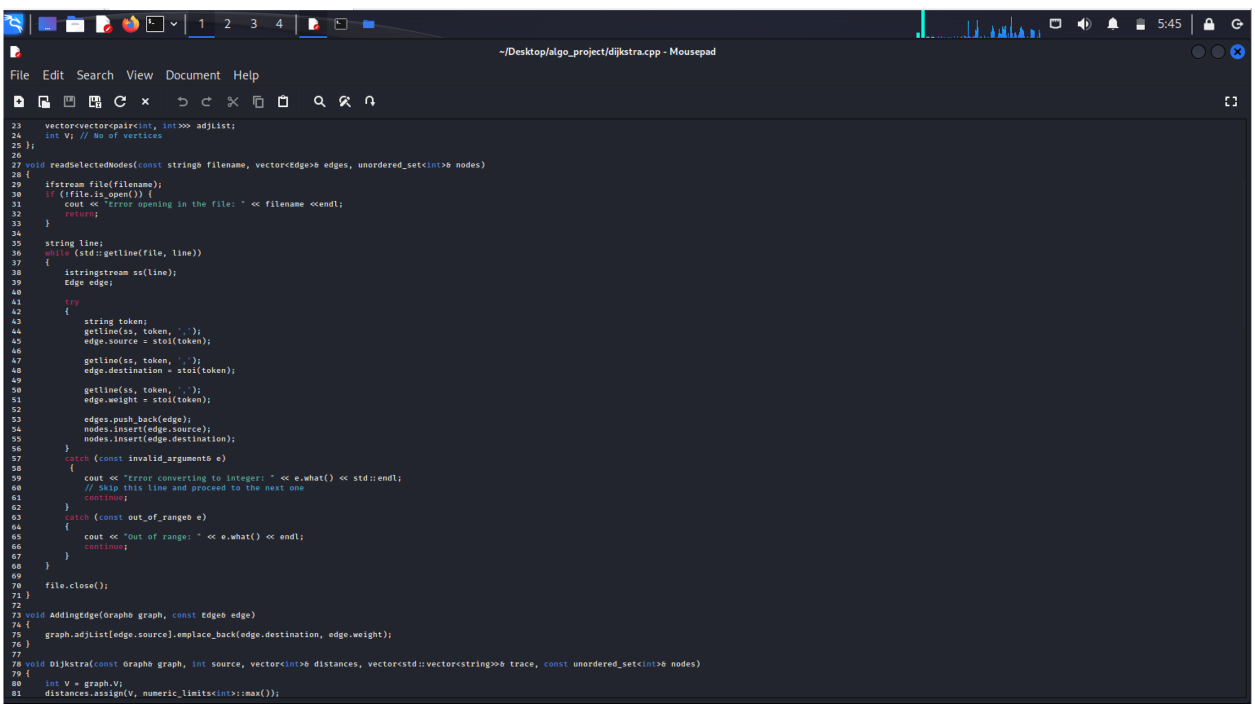
After running Dijkstra's algorithm, the code displays the shortest paths from a specified source node to all other nodes. It also writes these results, including the trace of the algorithm, to a CSV file named"dijkstra_shortest_distances_with_trace.csv".

- Conclusion:

Finally, the program outputs the execution time of Dijkstra's algorithm, providing insights into the efficiency of the implemented solution.

- Time Complexity of Dijkstra’s Algorithm:
  - Best Case: O((V + E) * log(V))
  - Average Case: O((V + E) * log(V))
  - Worst Case: O(V + E)

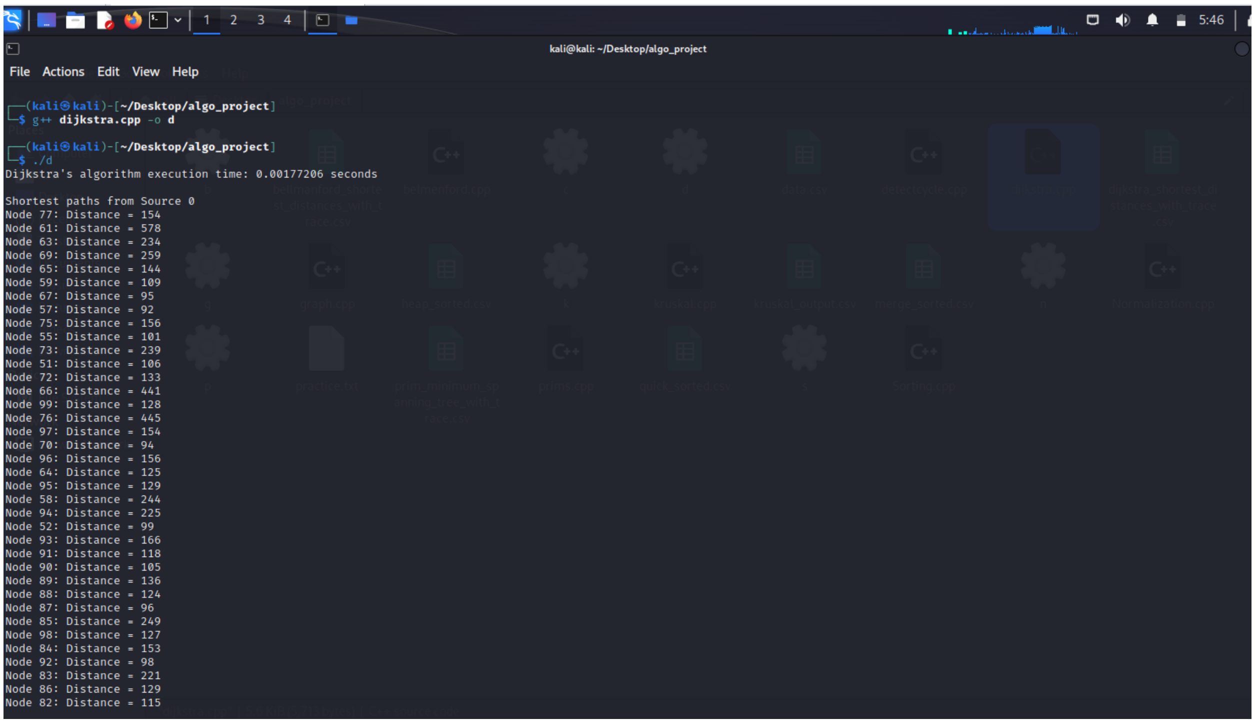
- Analysis:
- *Initialization (distances and traces):* O (V), Initializing the distance array and trace vector involves iterating over all vertices in the graph. Since the graph has V vertices, the time complexity is linear in the number of vertices.
- *Reading Input:* O (E), Reading edges from the file involves iterating over each edge once. In the worst case, we have E edges, leading to a linear time complexity in the number of edges.
- *Graph Initialization* (Adding Edges): O (E), Adding edges to the adjacency list involves iterating over each edge once. In the worst case, we have E edges, leading to a linear time complexity in the number of edges.
- *Dijkstra's Algorithm Main Loop:* O ((V + E) * log (V)), Dijkstra's algorithm with a binary heap priority queue has a worst-case time complexity of O ((V + E) * log (V)). This dominates the overall time complexity.
- *Writing Output:* O (V), Writing the results involves iterating over all vertices to write distances and traces. Since there are V vertices, the time complexity is linear in the number of vertices.
- *Overall Time Complexity: The* dominant Factor is Dijkstra's Algorithm resulting in a worst case time complexity of O ((V + E) * log (V)).




```cpp
    vector<vector<pair<int, int>>> adjList;
    int V; // No of vertices
};

void readSelectedNodes(const string& filename, vector<Edge>& edges, unordered_set<int>& nodes)
{
    ifstream file(filename);
    if (!file.is_open()) {
        cout << "Error opening in the file: " << filename <<endl;
        return;
    }

    string line;
    while (std::getline(file, line))
    {
        istringstream ss(line);
        Edge edge;

        try
        {
            string token;
            getline(ss, token, ',');
            edge.source = stoi(token);

            getline(ss, token, ',');
            edge.destination = stoi(token);

            getline(ss, token, ',');
            edge.weight = stoi(token);

            edges.push_back(edge);
            nodes.insert(edge.source);
            nodes.insert(edge.destination);
        }
        catch (const invalid_argument& e)
         {
            cout << "Error converting to integer: " << e.what() << std::endl;
            // Skip this line and proceed to the next one
            continue;
        }
        catch (const out_of_range& e)
        {
            cout << "Out of range: " << e.what() << endl;
            continue;
        }
    }

    file.close();
}

void AddingEdge(Graph& graph, const Edge& edge)
{
    graph.adjList[edge.source].emplace_back(edge.destination, edge.weight);
}

void Dijkstra(const Graph& graph, int source, vector<int>& distances, vector<std::vector<string>>& trace, const unordered_set<int>& nodes)
{
    int V = graph.V;
    distances.assign(V, numeric_limits<int>::max());
```

Output:

- Dijkstra Execution Time = 0.00177206



5. *BellmenFord:*

- **File Reading:**

  The code starts by reading a CSV file containing information about edges in a graph (source node, destination node, and edge weight).

- **Bellman-Ford Algorithm:**

  The Bellman-Ford algorithm is then applied to find the shortest paths from a given source node to all other nodes in the graph. The algorithm iterates over all edges multiple times, relaxing the edges by updating the distances whenever a shorter path is found.

- **Results Writing:**

  The results, including the shortest distances and the trace of the algorithm, are written to an output file.

- **Displaying Shortest Paths:**

  The code also prints the shortest paths from the source node to all other nodes on the terminal.

- Timing Measurement:

  The execution time of the Bellman-Ford algorithm is measured using <chrono> library. This helps in understanding how long the algorithm takes to run.

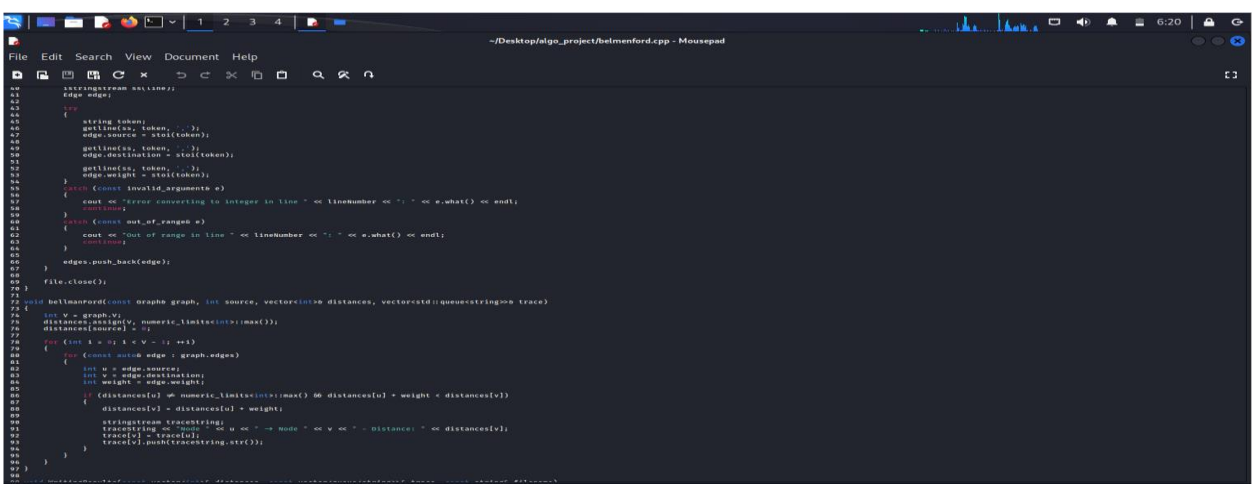
- Main Function:

  The main function run the entire process, from reading the file to applying the algorithm, measuring time, and writing the results.

- Time Complexity of Bellman-Ford Algorithm:
  - Best Case: O (V + E) – O (V)
  - Average Case: O (V * E)
  - Worst Case: O (V * E)

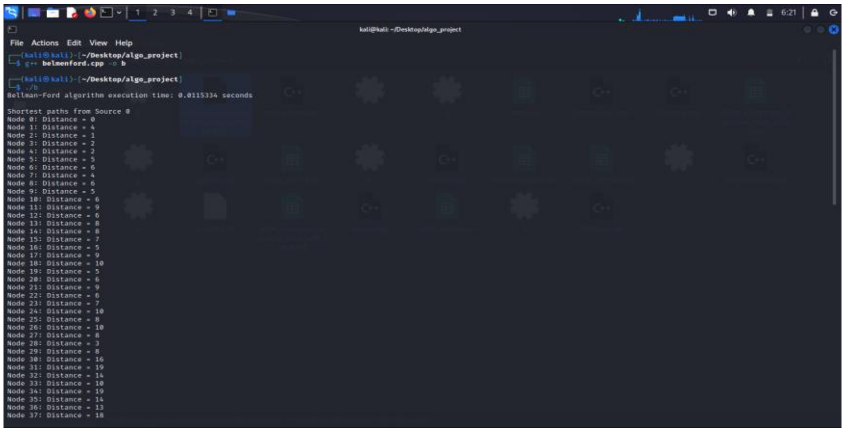
- Analysis:

- *Initialization (distances and traces):* O (V) Initializing the distance array and trace vector has a time complexity of O (V) because it involves iterating over all vertices.
- *Bellman-Ford Algorithm Main Loop:* O (V * E), the outer loop runs V-1 times, where V is the number of vertices. The inner loop iterates over all edges, which is E in the worst case (complete graph).
- *WritingResults Function:* O (V) Writing the results involves iterating over all vertices to write the distances and traces.
- *Overall Time Complexity:* The dominant factor is the Bellman-Ford main loop, resulting in a worst-case time complexity of O (V * E).



Output:

- BellmenFord Execution Time = 0.0115334



6. *Prim's:*

- **File Reading:**

  The program starts by reading data from a file named "data.csv" that represents a graph with nodes and weighted edges.

- **Graph Representation:**

  The program then creates a graph data structure based on the read data, where nodes are connected by edges, and each edge has a weight.

- **Prim's Algorithm:**

  Prim's algorithm is applied to find the Minimum Spanning Tree. It starts from an arbitrary node and systematically adds the smallest edge that connects a new node to the existing tree. The process continues until all nodes are included in the MST.

- **File Writing:**

  The results, including the MST and a trace of the algorithm's execution, are written to an output file name "prim_minimum_spanning_tree_with_trace.csv"

- **Display Function:**

  The program displays the MST on the terminal.

- Timing Measurement:

 The execution time of Prim's algorithm is measured and printed to the terminal.

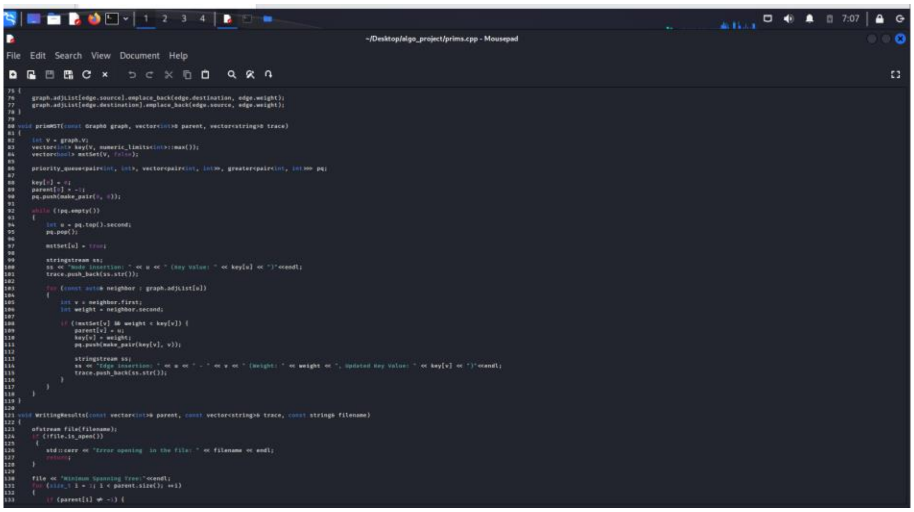
- Main Function:

 The main function run the entire process, from reading the file to applying Prim's algorithm, measuring time, displaying results, and writing them to an output file.

- Time Complexity of Bellman-Ford Algorithm:
  - Best Case: O(E.logV)
  - Average Case: O(E.logV)
  - Worst Case: O((V+E)·logV)

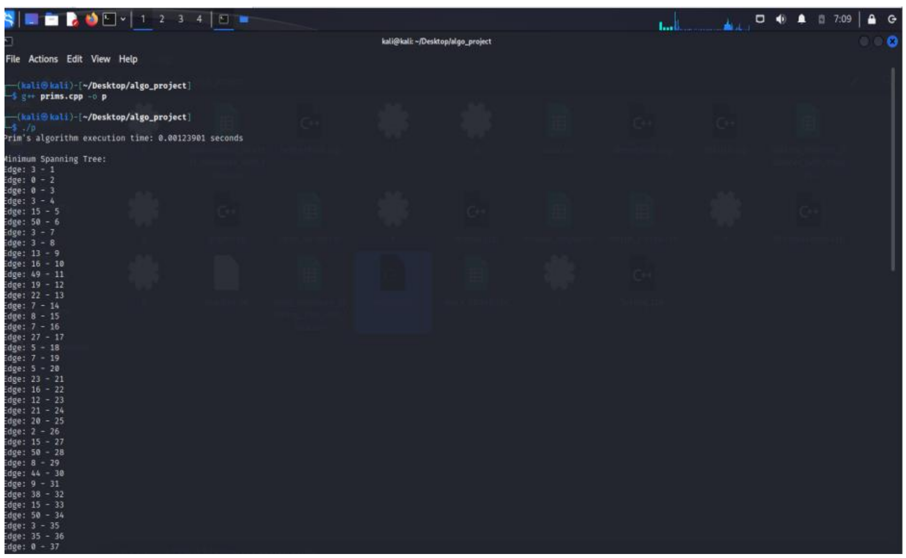
- Analysis:

- *Filereading Function:* O (E), where E is the number of edges. The function reads each line from the file, and in the worst case, it reads all the edges.
- *AddingEdge Function:* O (1) (constant time). The function appends the edge to the adjacency list.
- *primMST Function:* O (E + V log V), where E is the number of edges and V is the number of vertices. This is due to the priority queue operations inside the while loop.
- *WritingResults Function:* O (V), where V is the number of vertices. The function iterates over all vertices to write the results.
- *displayMST Function:* O (V), where V is the number of vertices. Similar to WritingResults, it iterates over all vertices.
- *Timemeasurement Function:* Depends on the function passed to it. In this context, it's used to measure the execution time of primMST, so its complexity is O (E + V log V).
- *Main Function:* O (E + V log V), dominated by the call to primMST.
- *Overall Time complexity:* The overall time complexity is determined by the primMST function, making it O (E + V log V). The initialization and other operations contribute less to the overall complexity.



Output:

- Prims' Execution Time = 0.00123901



7. *Kruskal:*

- **File Reading Function:**

This function reads data from a CSV file containing information about edges in a graph. Each line of the file represents an edge with source node, destination node, and weight. The function handles potential errors while reading the file, such as invalid data or out-of-range values.

- Edge Structure:

  The code defines a structure (Edge) to represent an edge in the graph. It stores information about the source node, destination node, and the weight of the edge.

- Disjoint Set Data Structure:

  This structure is used to implement the disjoint-set data structure, which helps in efficiently detecting and merging connected components in a graph.

- Kruskal's Algorithm:

  The main algorithm implemented in the code is Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a graph. The algorithm iteratively selects edges in ascending order of their weights, adding them to the MST if they don't create a cycle. The progress of the algorithm and its decisions (acceptance or rejection of edges) are recorded in a trace.

- Time Measurement Function:

  This function measures the execution time of a given function using <chrono> library. In this case, it measures the time taken by Kruskal's algorithm to find the MST.

- Main Function:

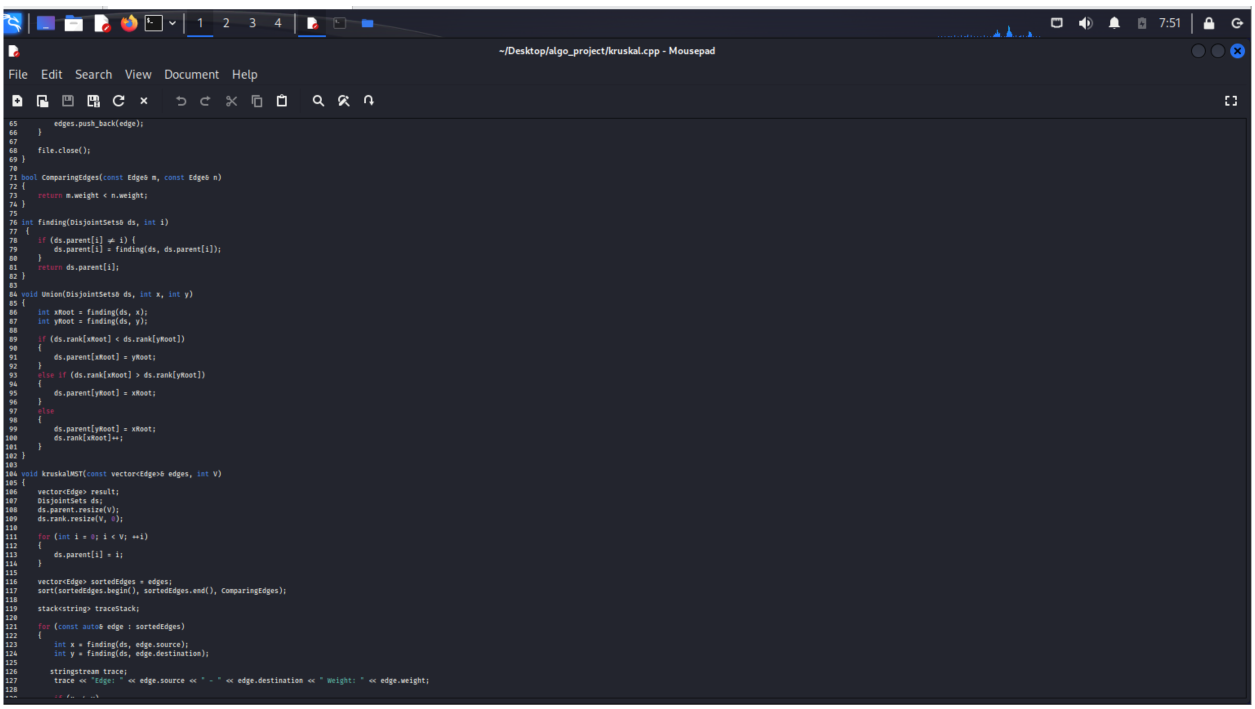
  Reads data from the CSV file using the Filereading function. Determines the number of nodes in the graph. Applies Kruskal's algorithm to find the MST and records the execution time. Outputs the MST and a trace of the algorithm's steps to a CSV file (kruskal_output.csv).Outputs the MST to the terminal. Outputs the execution time of Kruskal's algorithm.

- Time Complexity of Kruskal's Algorithm:
  - Best Case: O(E log E)
  - Average Case: O(E log E)
  - Worst Case: O(E log E)

- Analysis:

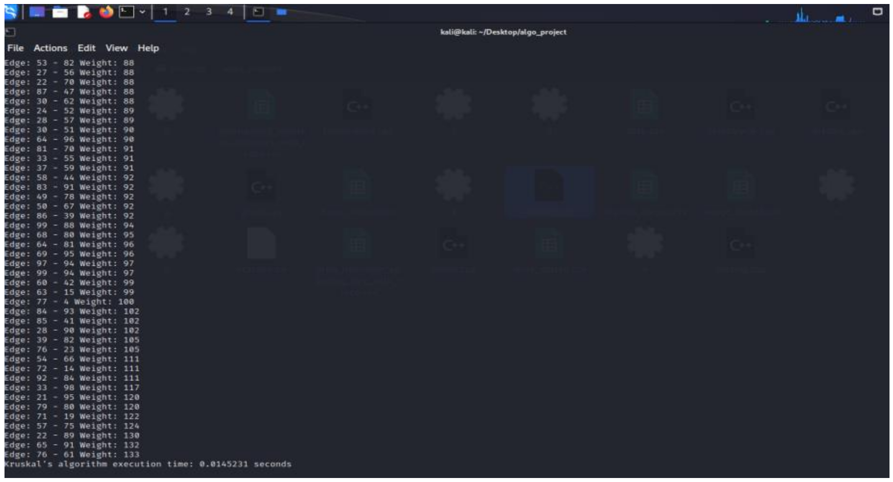
- *File Reading:* O (E), where E is the number of edges. This is because each edge needs to be processed once while reading from the file.
- *Kruskal's Algorithm:* O (E log E), where E is the number of edges. This is the dominant factor and represents the time taken to sort the edges.

- *Disjoint Set Operations:* O (E log* V), where V is the number of vertices. The log* function is the iterated logarithm and grows extremely slowly.
- *Constructing Minimum Spanning Tree:* O(E log* V)
- *Writing Results to Output File:* O (E) for writing the MST and trace to the output file.
- *Time Measurement:* O (T), where T is the time taken by the provided function kruskalMST.
- *Overall Time Complexity: The d*ominant factor is sorting edges (O (E log E)), resulting in a time complexity of O (E log E).



Output:

- Kruskal's Execution time = 0.0145231



8. *DetectCycle:*

- **Edge Structure:**

  Represents a connection between two points (vertices). Each connection has a starting point (source), an ending point (destination), and a weight.

- **File Reading:**

  Reads information about connections from a file. Each line in the file represents a connection and contains details such as the starting point, ending point, and the cost of the connection.

- **Sorting Edges:**

  Sorts the connections based on their costs in ascending order.

- **Union-Find Operations:**

  Implements a method to check for loops in the connections without drawing the entire map. Uses a technique called union-find to keep track of connected points. If connecting two points would create a loop, it is detected during this process.

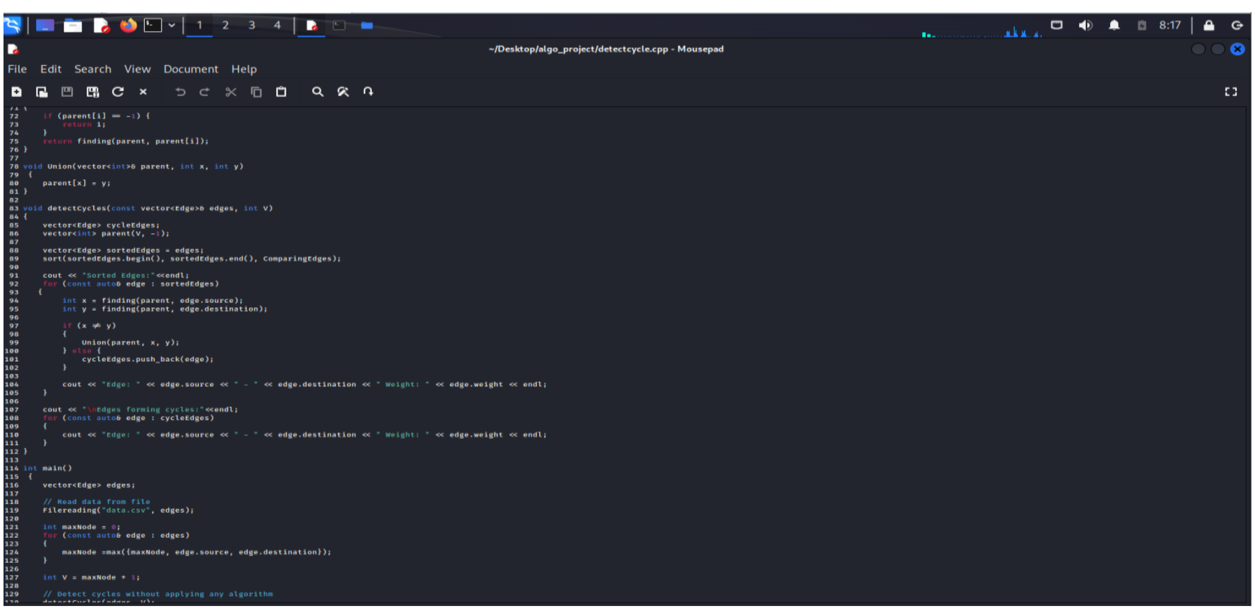
- **Detecting Cycles:**

  Outputs the connections that would create loops in the network. Highlights specific connections that, if taken, would form a loop.

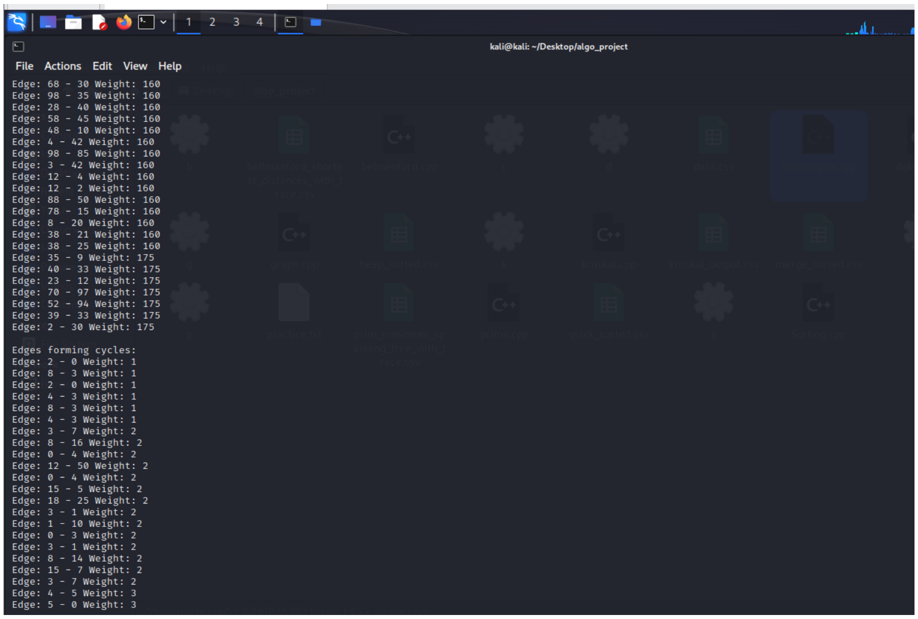
- ## Time Complexity of Detect Cycle Algorithm:
  - Best Case: O(E)
  - Average Case: O(E log E)
  - Worst Case: O(E log E + E) or O(E log E + E log V)

- ## Analysis:

- *File Reading:* O (E), the function reads the edges from a file line by line and processes each edge once. The time complexity is linear with respect to the number of edges.
- *Sorting Edges:* (E log E), the edges are sorted based on their weights using the sort function. This is the dominant factor in the overall time complexity. Sorting has a time complexity of O (E log E).
- *Disjoint Set Operations:* O (E log* V), the disjoint set operations involve finding the representative of a set (finding) and union operation. The time complexity is determined by the union-find operations during the Minimum Spanning Tree construction. The log* V term accounts for the efficiency of disjoint set operations.
- *Detecting Cycles:* O (E log E) the detect Cycles function utilizes sorting and disjoint set operations to find cycles in the graph.
- *Writing Results to Output File:* O (E) The output of sorted edges and edges forming cycles involves printing each edge once.
- *Overall Time Complexity:* The overall time complexity is dominated by the sorting of edges. Resulting time complexity of O (E log E).

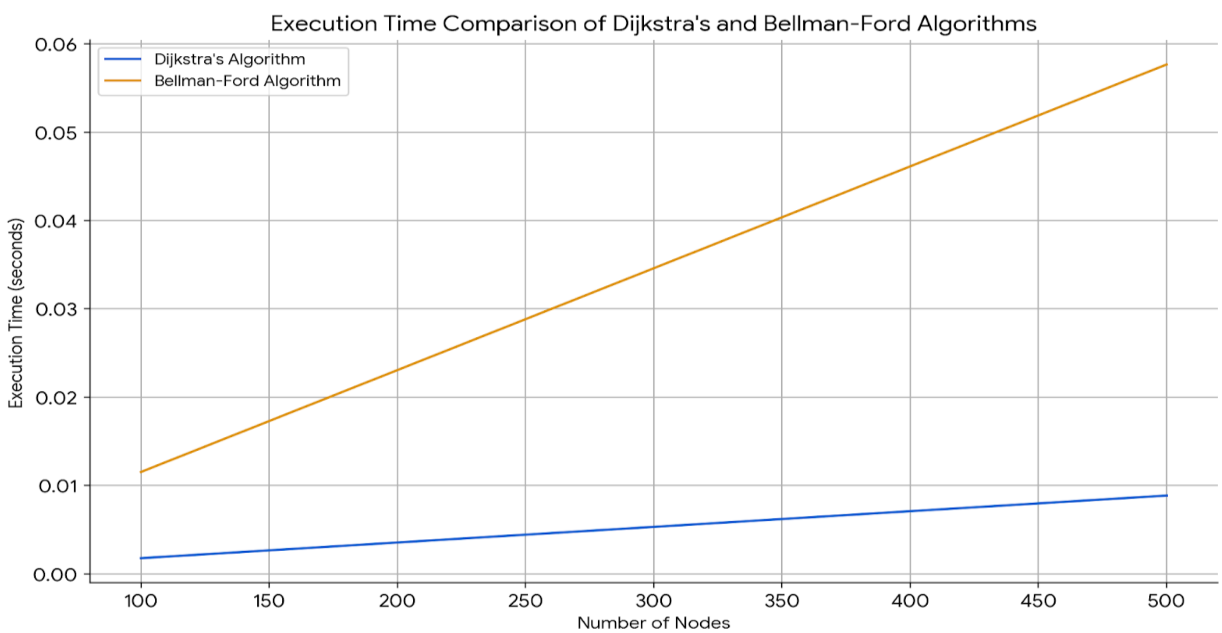


__Output:__

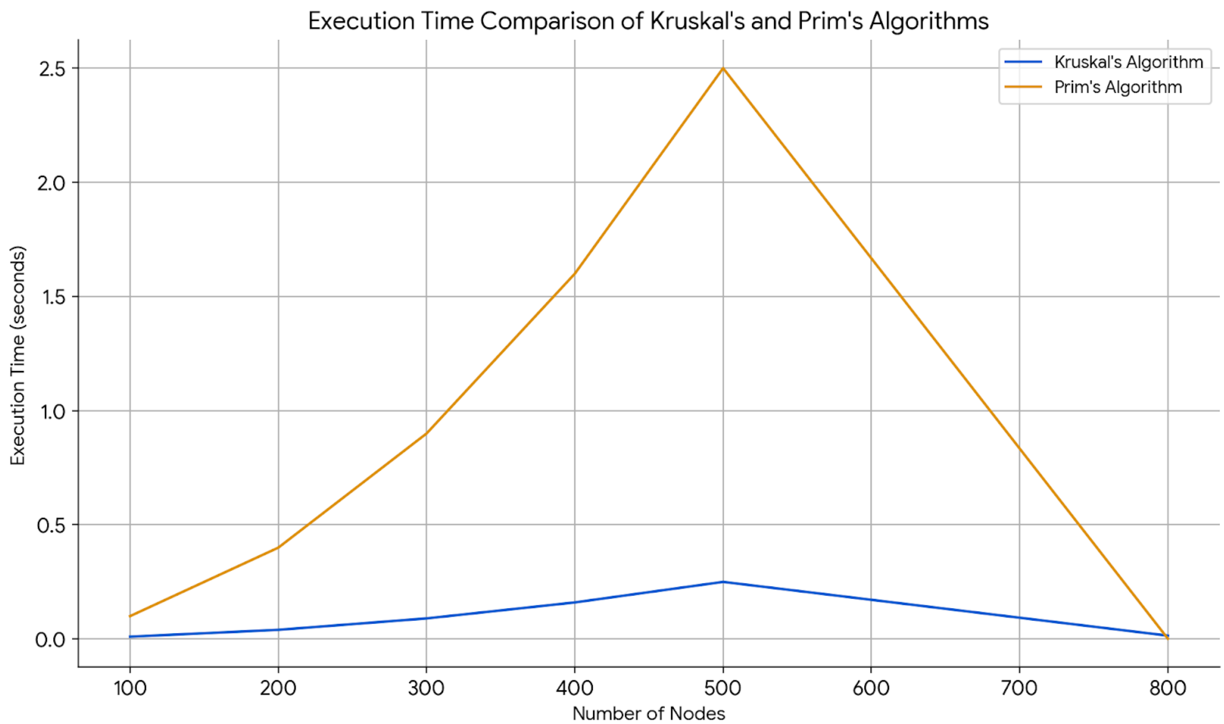


__GRAPHICAL COMPARISON__

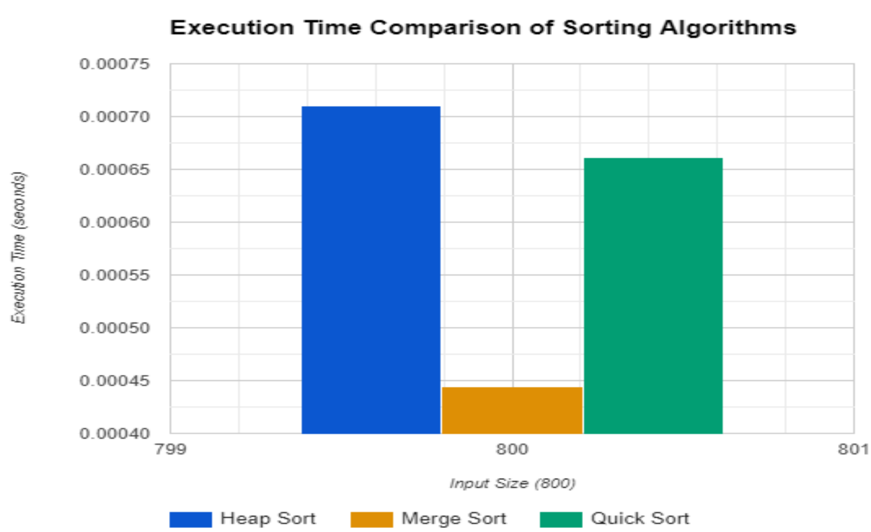
Dijkstra and BellmenFord:



Prims and Kruskal:



Merge Sort, Heap Sort and Quick Sort:



Implementation structure:

- Dijkstra:

In Dijkstra code, stack was not used but priority queue is used. Using of queues is crucial in Dijkstra as it heavily relies on selecting and relaxing edges based on their weights. Absence of queues to handle vertex selection can degrade the algorithm’s efficiency. In addition, we used priority queues that decreases the time complexity of the code. With a simple queue, the time complexity of Dijkstra's algorithm using an adjacency list is O (V^2) where V is the number of vertices. Using a priority queue the time complexity reduces to O ((V + E) * log V), where E is the number of edges. This improvement arises from the faster extraction of minimum-distance vertices.

- Bellman-Ford:

In the Bellman-Ford algorithm, queues are used to trace the path of the shortest distance from the source node to each node in the graph. However, a stack is used along with the queue to help optimize the code in certain scenarios, especially when dealing with the tracing of paths.

- Prims:

Prim's algorithm employs a priority queue to efficiently select the minimum-weight edge in each iteration. The priority queue stores pairs of vertices and their corresponding edge weights, facilitating the selection of the minimum-weight edge efficiently. The optimization in this code primarily lies in the utilization of the priority queue data structure within the context of Prim's algorithm, thereby contributing to the overall time complexity optimization.

- Kruskal’s:

In the Kruskal's algorithm implementation there isn't explicit uses of queues or stacks for optimizing the time complexity. Instead, the optimization lies in the algorithmic approach itself. Kruskal's algorithm operates by sorting the edges based on their weights and then greedily selecting the smallest weighted edges that do not form a cycle in the Minimum Spanning Tree (MST). Kruskal's algorithm uses a disjoint set data structure with the "Union" and "Find" operations. These operations have nearly constant time complexity in practical scenarios, amortized to nearly O (1) per operation, leading to efficient merging of sets and finding connected components. The stack is used to keep track of the accepted or rejected edges along with their details. However, its primary purpose here is for creating the trace output rather than optimizing the algorithm's time complexity.

Conclusion:

In summary, this comprehensive analysis of various algorithms and data structures showcases their diverse applications and optimizations. Dijkstra's algorithm, leveraging priority queues, excels in efficiently finding shortest paths in weighted graphs. Bellman-Ford algorithm, employing queues and stacks, stands resilient in handling scenarios with negative-weight edges. Prims' and Kruskal's algorithms, utilizing priority queues and disjoint sets, respectively, demonstrate their prowess in finding minimum spanning trees with efficiency. The sorting algorithms—Merge Sort, Heap Sort, and Quick Sort—highlight their varying time complexities, with Merge Sort standing out for its stability. The implementation structure emphasizes the significance of queues, priority queues, and disjoint sets in optimizing algorithmic efficiency. Overall, we underscores the importance of choosing the right algorithm and data structure for specific tasks, considering their time complexities and application contexts.

THE END…..