

Algorithm Design & Analysis Group Assignment

Prepared by Erfan Rahmani

Num	Student Name	Student ID	Task Description	Percentage
1	Rahmani Erfan	X	Everything	100%

Huffman Input

Code Structure & Explanation

Libraries & Usages

```
C/C++  
#include <iostream>  
#include <fstream>  
#include <vector>  
#include <random>  
#include <string>  
#include <algorithm>
```

❖ **<vector>**: It's used to keep track of the unique characters generated.

- ❖ **<random>**: This library includes various random number generators. I have used the Mersenne Twister algorithm. The Mersenne Twister code is used to initialise a random number generator engine (std::mt19937) with a seed derived from the current time & my ID. This engine is then used along with a uniform distribution (std::uniform_int_distribution<>) to generate random numbers for selecting characters and determining the length of words.

Generating Random Words

```
C/C++
std::string generateRandomWord() {
    static const std::string chars = "abcdefghijklmnopqrstuvwxyz";
    std::random_device rd;
    unsigned seed = rd() ^ 1201102372;
    std::mt19937 gen(seed);
    std::uniform_int_distribution<> dis(0, chars.size() - 1);

    int length = (dis(gen) % 2) + 4; // Randomly choose length 4 or 5
    (length of word)
    std::string word;

    for (int i = 0; i < length; ++i) {
        word += chars[dis(gen)];
    }

    return word;
}
```

- ❖ This function generates a random word of length 4 or 5 using lowercase alphabetic characters by using the random number generator (Mersenne Twister). It selects characters from a predefined set of lowercase alphabets and constructs the word by appending them together.

Huffman Input Main Function

```
C/C++
```

```

int main() {
    int numWords;

    // Set the seed for random number generation
    std::random_device rd;
    unsigned seed = rd() ^ 1201102372; //bitwise XOR between random device
and the seed. (If seed is used only, output will all be the same)
    std::mt19937 gen(seed); //Generator Engine
    std::uniform_int_distribution<> dis(1,26);
    int randUniqueChars = dis(gen);

    std::cout << "Enter the number of words to generate: ";
    std::cin >> numWords;

    // Generate unique characters
    std::vector<char> uniqueChars;

    for (int i = 0; i < randUniqueChars; ++i) {
        std::string word = generateRandomWord();

        for (char c : word) { //iterate through every character in word.
            if (std::find(uniqueChars.begin(), uniqueChars.end(), c) ==
uniqueChars.end()) {
                uniqueChars.push_back(c); //appending the unique characters.
            }
        }
    }

    // Write the output to a text file
    std::ofstream outputFile("huffmancoding_" + std::to_string(numWords) +
"_input.txt");
    if (!outputFile) {
        std::cerr << "Failed to open the output file." << std::endl;
        return 1;
    }

    // Write the number of unique characters
    int numUniqueChars = uniqueChars.size();
    outputFile << numUniqueChars << std::endl;

    // Write the list of unique characters
    for (char c : uniqueChars) {
        outputFile << c << "\n";
    }
    outputFile << std::endl;

    // Write the list of words

```

```

for (int i = 0; i < numWords; ++i) {
    outputFile << generateRandomWord() << " ";
}
outputFile << std::endl;

outputFile.close();

std::cout << "Complete!" << std::endl;

return 0;
}

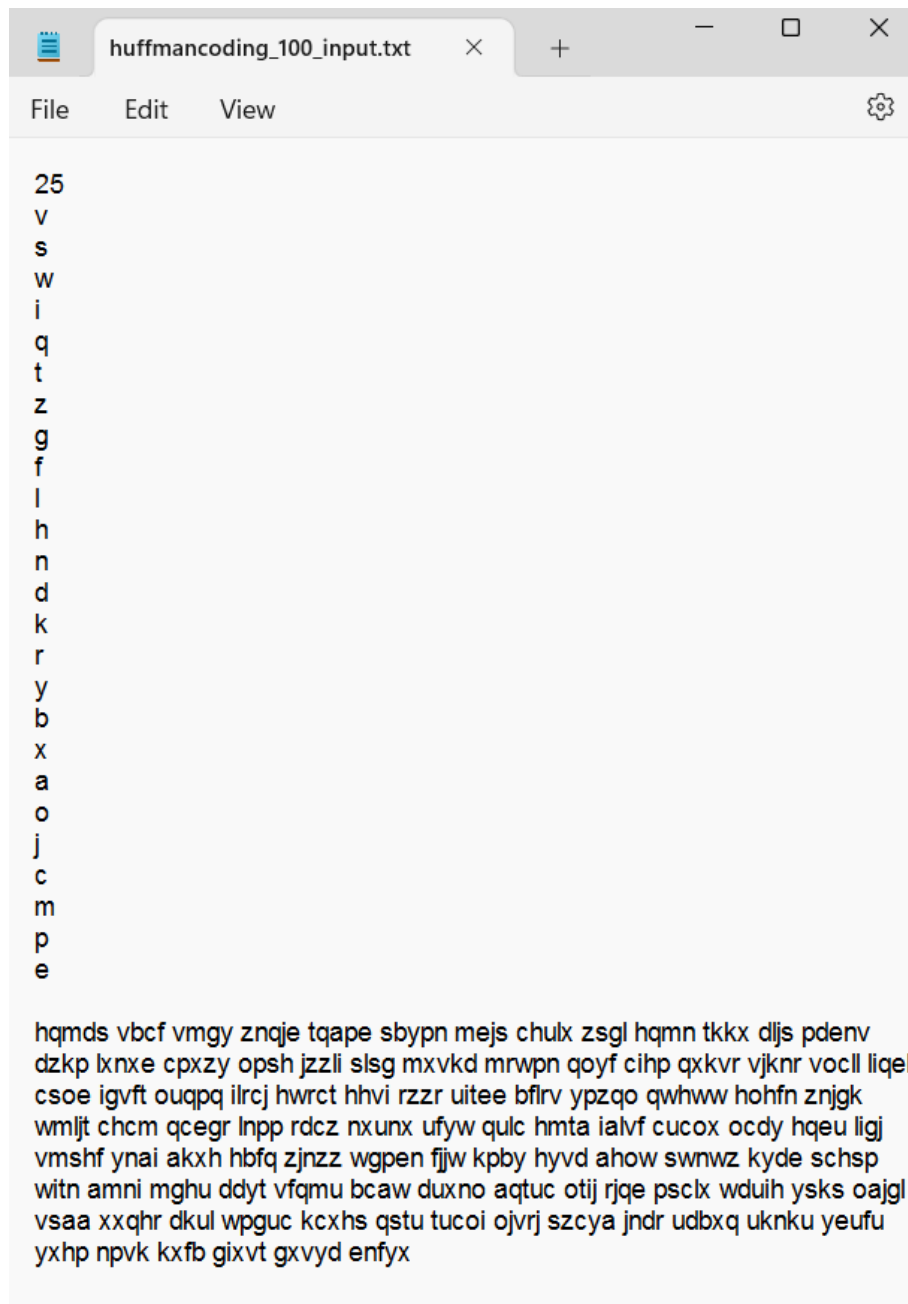
```

❖ Flow of Main function:

- Set the seed for random number generation (for number of unique characters).
- Ask the user for the number of words to generate.
- Generate unique characters.
- Write the number of unique characters to the file.
- Write the list of unique characters to the file.
- Write the list of generated words to the file.
- Close the output file.
- Print "Complete!" to the console.

Output Analysis

Screenshot Sample



The screenshot shows a text editor window titled "huffmancoding_100_input.txt". The window has a menu bar with "File", "Edit", and "View" options, and a settings gear icon. The text content is as follows:

```
25
v
s
w
i
q
t
z
g
f
l
h
n
d
k
r
y
b
x
a
o
j
c
m
p
e

hqmds vbcf vmgy znqje tqape sbypn mejs chulx zsgl hqmn tkkx dljs pdenv
dzkp lxnxe cpxzy opsh jzzli slsg mxvkd mrwpn qoyf cihp qxkvr vjknr vocll liqel
csoe igvft ouqpq ilrcj hwrct hhvi rzzr uitee bflrv ypzqo qwhww hohfn znjgk
wmljt chcm qcegr lnpd rdcz nxunx ufyw qulc hmta ialvf cucox ocody hqeu ligj
vmshf ynai akxh hbfq zjnzz wgpen fjfw kpby hyvd ahow swnwz kyde schsp
witn amni mghu ddyt vfgmu bcaw duxno aqtuc otij rjqe pscix wduih ysks oajgl
vsaa xxqhr dkul wpguc kcxhs qstu tucoi ojvrj szcya jndr udbxq uknku yeufu
yxhp npvk kxfo gixvt gxvyd enfyx
```

Huffman Output

Code Structure & Explanation

Libraries & Usages

```
C/C++  
#include <iostream>  
#include <fstream>  
#include <vector>  
#include <map>  
#include <queue>  
#include <bitset>  
#include <chrono>  
#include <string>  
#include <iomanip>  
#include <algorithm>
```

- ❖ **<map>**: This library allows the program to create a mapping between characters and their frequencies, which is useful for tracking how often each character appears in the text.
- ❖ **<queue>**: It provides a queue data structure that allows efficient processing of elements based on their priorities, which is used here to build the Huffman tree.
- ❖ **<bitset>**: It provides a way to manipulate sequences of bits, but it's not directly used in this specific code.
- ❖ **<chrono>**: Used for measuring time durations. It is used to calculate the execution time of the program, which is then written to the output file.
- ❖ **<string>**: It provides various operations for working with strings, such as storing compressed text and generating Huffman codes.
- ❖ **<iomanip>**: Used for formatting output. It is used to set the decimal place of the time taken, which is then written to the output file.

Huffman Tree Node Structure

```
C/C++
struct Node {
    char character;
    int frequency;
    Node* left;
    Node* right;

    Node(char ch, int freq) : character(ch), frequency(freq), left(nullptr),
right(nullptr) {}
};
```

The purpose of this structure is to store the nodes for the Huffman tree. As shown, each node has variables:

- Character: The character in the node
- Frequency: Frequency of the character.
- Node* left: Left node to current node.
- Node* right: Right node to current node.

The constructor of the structure takes two variables character & frequency, it also initially sets the left and right to `nullptr`.

Generating Codes for the Characters

```
C/C++
void generateCodes(Node* root, std::map<char, std::string>& codes,
std::string code) { // map to store code.
    if (root == nullptr)
        return;

    //checking if current node is a leaf node.
    if (root->left == nullptr && root->right == nullptr) {
        codes[root->character] = code;
    }

    generateCodes(root->left, codes, code + "0");
    generateCodes(root->right, codes, code + "1");
}
```

The code generator is a double recursive function that calls itself until the last root node is reached.

- ❖ Components of the function:
 - The map is used to map the characters to the codewords.
 - If both the left and right nodes of the current nodes are `nullptr`, a leaf node is reached. Therefore, the current root will store the character.
 - By the time the function is completed, the code map will contain all of the code words for the characters.

Generating Huffman Tree

```
C/C++
// Building huffman tree using pq.
Node* buildHuffmanTree(const std::map<char, int>& frequencyMap) {
    std::priority_queue<Node*, std::vector<Node*>, Compare> pq; // type
    node, contained in a vecot with a custom comparator ( declared above)

    // Storing each character-frequency pairs in a node.
    for (const auto& pair : frequencyMap) {
        pq.push(new Node(pair.first, pair.second));
    }

    // Iterating through pq until one node left.
    while (pq.size() > 1) {
        //two lowest frequencies.
        Node* left = pq.top();
        pq.pop();
        Node* right = pq.top();
        pq.pop();
        //creating parent node for the nodes.
        Node* parent = new Node('$', left->frequency + right->frequency);
        //parent node has a sum of children freq.
        parent->left = left;
        parent->right = right;

        //Parent merged with nodes.
        pq.push(parent);
    }
    //root of pq.
    return pq.top();
}
```

The generating of codes for the characters follows this flow:

A Huffman tree is built using a priority queue. The function takes a frequency map as its input.

Function flow:

- ❖ Create a priority queue (pq) using a [custom comparator](#) to store nodes.
- ❖ Iterate through the character-frequency pairs in the given frequencyMap.
 - Create a new node for each pair and push it into the priority queue.
- ❖ While the priority queue has more than one node:
 - Extract the two nodes with the lowest frequencies from the top of the priority queue.
 - Create a new parent node with the character '\$' and the sum of frequencies of the two extracted nodes.
 - Assign the extracted nodes as the left and right children of the parent node.
 - Push the parent node back into the priority queue.
 - Repeat until the pq has one node left.
- ❖ Return the root node of the priority queue, which represents the Huffman tree.

Custom Comparator

```
C/C++
struct Compare {
    bool operator()(Node* lhs, Node* rhs) const {
        return lhs->frequency > rhs->frequency;
    }
};
```

The purpose of this structure is to be used by the **priority queue** in the [huffman tree generation](#).

It compares the lhs (left hand side) to the rhs (right hand side) frequencies.

Compressed percentage and stats

```
C/C++
void calculateCompressionStats(const std::string& originalText, const
std::string& compressedText, int& totalBits, double& spacePercentage) {
    totalBits = originalText.size() * 8;
    int compressedBits = compressedText.size();
    spacePercentage = (1 - static_cast<double>(compressedBits) / totalBits)
* 100.0;
}
```

- ❖ This Function takes two string inputs: originalText and compressedText.

- It calculates the total number of bits in originalText and assigns it to totalBits (8 bits per character, which is just an assumption throughout this program).
- It calculates the number of bits in compressedText and assigns it to compressedBits.
- It calculates the space saved percentage.
- The result is stored in spacePercentage.

Huffman Output Main Function

```
C/C++
    std::chrono::duration<double> duration = end - start;
    outputFile << "\n" << std::fixed << std::setprecision(5) <<
duration.count() << "s" << std::endl;

    outputFile.close();

    std::cout << "Compression complete!" << std::endl;

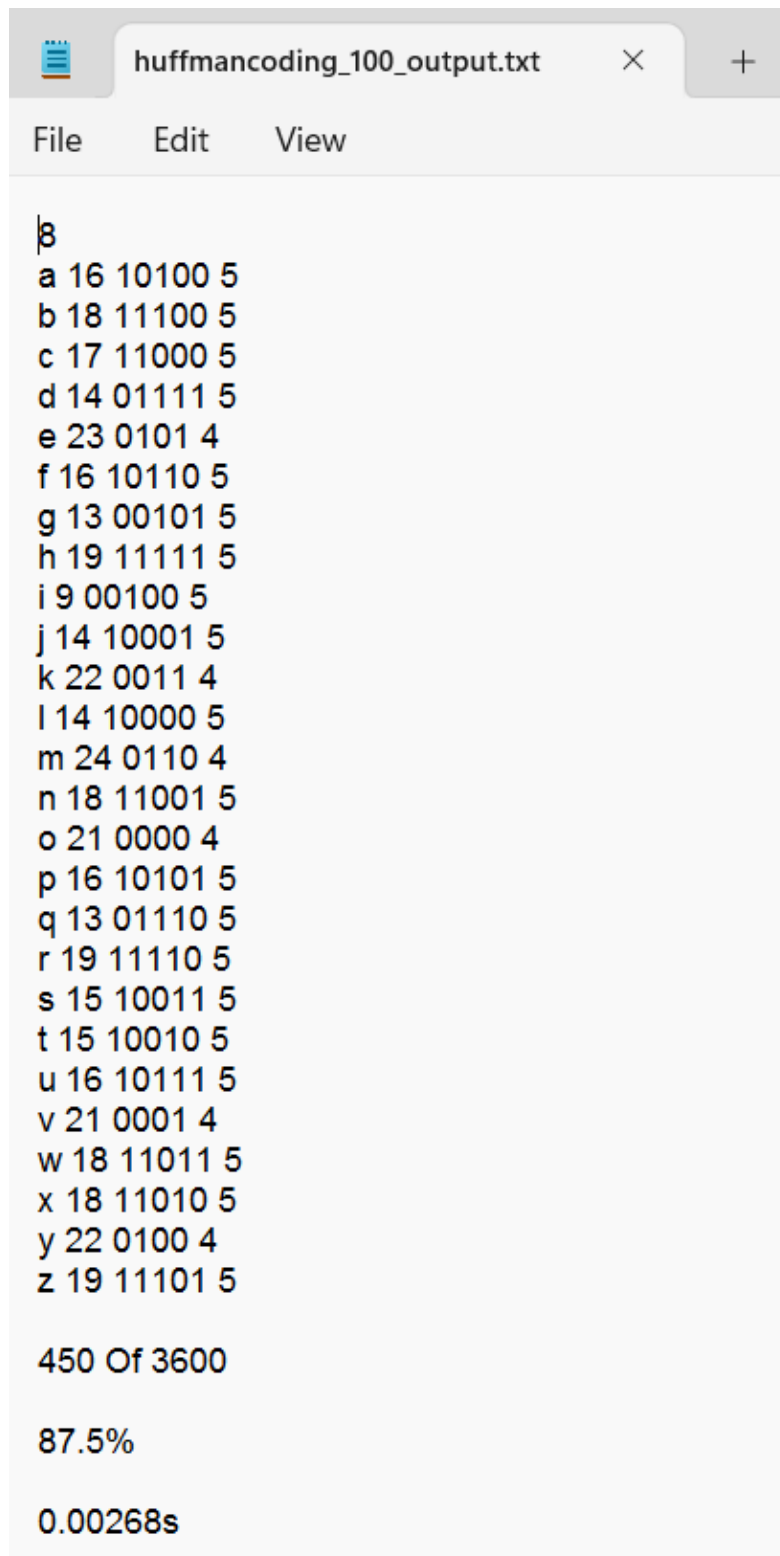
    return 0;
}
```

❖ Main function flow:

- Prompt user for the number of words.
 - This is used to read the file based on the number of words. Since the input file contains the number of words in the file.
- Start measuring execution time.
- Open input file and check if it fails.
- Read number of unique characters.
- Read unique characters from the input file.
- Read words and concatenate them into "compressedText".
- Close input file.
- Count character frequencies in "compressedText".
- [Build Huffman tree.](#)
- [Generate Huffman codes.](#)
- [Calculate total bits and compression space percentage.](#)
- Open the output file and check if it fails.
- Write number of unique characters to output file.
- Write character frequencies, codes, and bit counts to output files.
- Write total bits to the output file.
- Write compression space percentage to output file.
- Measure execution time.
- Close output file.
- Print "Compression complete!" to console.

Output Analysis

Screenshot Sample



The screenshot shows a text editor window titled "huffmancoding_100_output.txt". The editor displays a list of characters from 'a' to 'z' and a space character, each followed by a Huffman code and a frequency value. The output is as follows:

```

|
a 16 10100 5
b 18 11100 5
c 17 11000 5
d 14 01111 5
e 23 0101 4
f 16 10110 5
g 13 00101 5
h 19 11111 5
i 9 00100 5
j 14 10001 5
k 22 0011 4
l 14 10000 5
m 24 0110 4
n 18 11001 5
o 21 0000 4
p 16 10101 5
q 13 01110 5
r 19 11110 5
s 15 10011 5
t 15 10010 5
u 16 10111 5
v 21 0001 4
w 18 11011 5
x 18 11010 5
y 22 0100 4
z 19 11101 5

450 Of 3600

87.5%

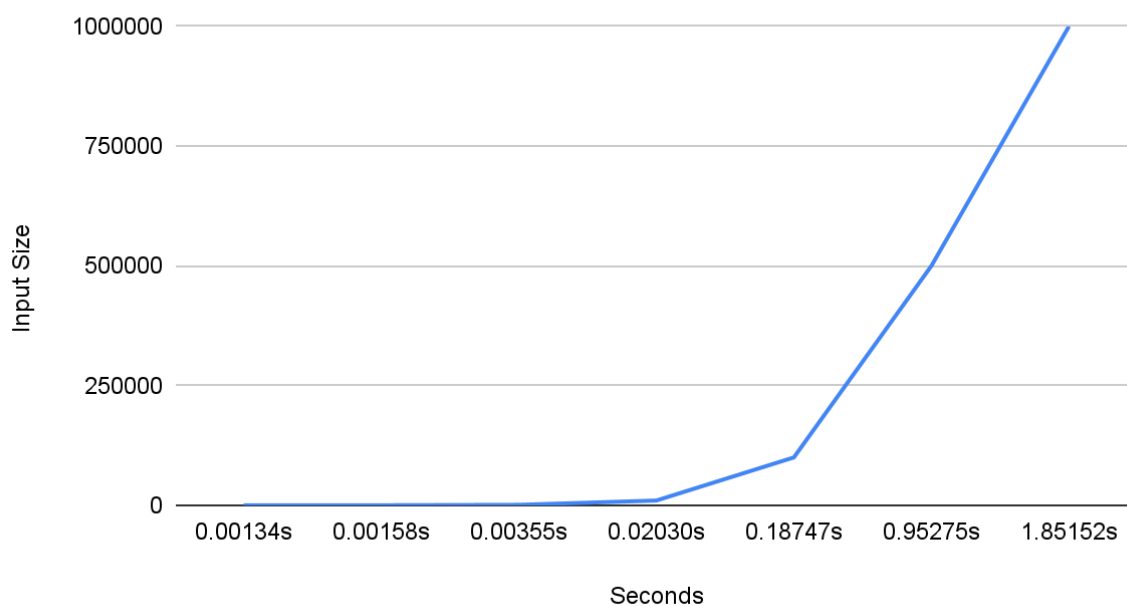
0.00268s
```

Huffman - Comparative Analysis

Input Size VS Time Findings:

Seconds	Input Size
0.00134s	10
0.00158s	100
0.00355s	1000
0.02030s	10000
0.18747s	100000
0.95275s	500000
1.85152s	1000000

Huffman Coding - Input Size vs Seconds

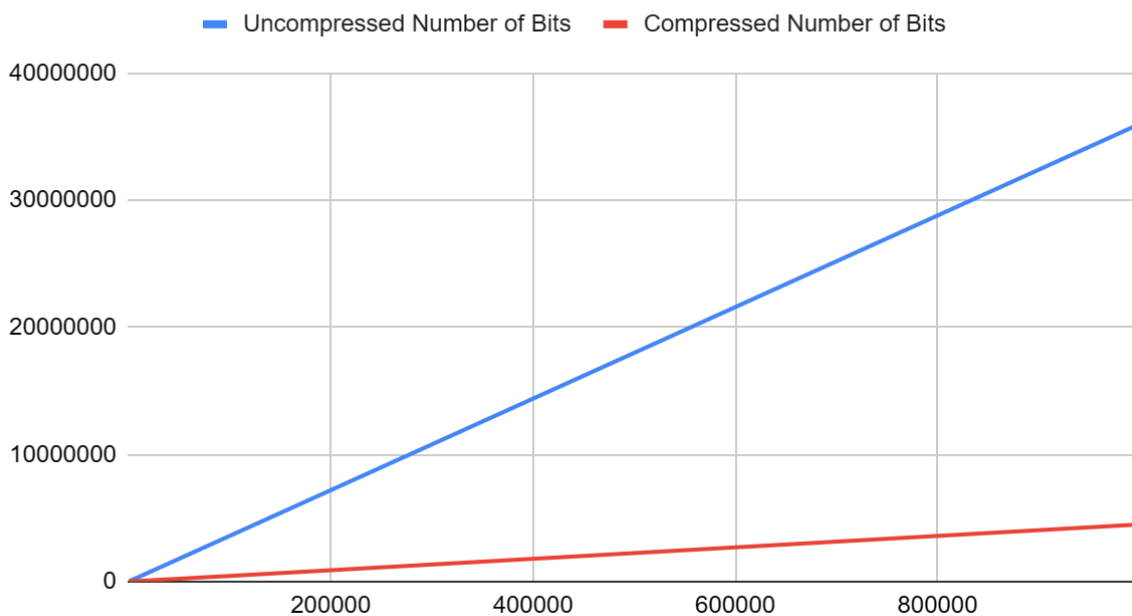


The time complexity of the output algorithm is **$O(n \log n)$** , where n is the number of inputs. This complexity arises from building the Huffman tree and generating the Huffman codes. As the number of characters increases, the time required to compress the input grows at a rate proportional to the logarithm of the number of characters.

Uncompressed VS Compressed Findings:

Compressed Number of Bits	Uncompressed Number of Bits	Input Size
45	360	10
463	3704	100
4496	35968	1000
44973	359784	10000
449836	3598688	100000
2249625	17997000	500000
4500479	36003832	1000000

Bit Size(Compressed & Uncompressed) and Input Size



The program efficiently applies Huffman coding to compress the input text, resulting in a substantial decrease in the number of required bits. As the input size grows, the compression ratio improves.

Kruskal's Algorithm Input

Code Structure & Explanation

Libraries & Usages

```
C/C++
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <random>
#include <algorithm>
```

- ❖ **<random>**: This library includes various random number generators. I have used the Mersenne Twister algorithm. The Mersenne Twister code is used to initialise a random number generator engine (`std::mt19937`) with a seed derived from the current time & my ID. This engine is then used along with a uniform distribution (`std::uniform_int_distribution<>`) to generate random numbers for selecting characters and determining the length of words.

Generating Vertex Names

```
C/C++
std::string generateVertexName(int index) {
    std::string name;
    while (index >= 0) {
        name.insert(name.begin(), 'A' + (index % 26));
        index /= 26;
        index--;
    }
    return name;
}
```

This function takes an integer index as input and returns a string representing a vertex name.

It uses a while loop to construct the name by repeatedly dividing the index by 26 and inserting the corresponding character ('A' plus the remainder) at the beginning of the string. The loop continues until the index becomes negative, and then the constructed name is returned as the result.

Essentially, this function generates vertex names in a sequence starting from 'A', 'B', ..., 'Z', 'AA', 'AB', ..., 'ZZ', 'AAA', and so on.

Generating Adjacency Matrix

C/C++

```
// Adjacency matrix generation
std::vector<std::vector<std::string>> generateAdjacencyMatrix(int
numVertices) {
    std::vector<std::vector<std::string>> matrix(numVertices,
std::vector<std::string>(numVertices));

    std::random_device rd;
    unsigned seed = rd() ^ 1201102372;
    std::mt19937 gen(seed); //Generator Engine
    std::uniform_int_distribution<> dis(0,1); //To decide whether connection
is made or not
    std::uniform_int_distribution<> weightDis(1, 10); // For edge weights

    for (int i = 0; i < numVertices; i++) {
        for (int j = i; j < numVertices; j++) {
            if (i == j) {
                matrix[i][j] = "i"; // Vertex not connected to itself
            }
            else {
                int isConnected = dis(gen); // deciding whether vertices
will be connected or not
                if (isConnected) {
                    int weight = weightDis(gen); // Random weight from 1 to
10

                    matrix[i][j] = std::to_string(weight);
                    matrix[j][i] = std::to_string(weight); // Making sure
the adjacency matrix is symmetric
                }
                else {
                    matrix[i][j] = "i";
                    matrix[j][i] = "i";
                }
            }
        }
    }

    return matrix;
}
```

The function creates a 2D vector representing an adjacency matrix based on the input number of vertices.

Inside the function, a random number generator is initialised, and two uniform distributions (from the random number generator) are set up to determine connections and edge weights. One distribution is to decide whether the edges are connected (0 OR 1), the other is used to decide the weights between connected edges.

The function uses nested loops to iterate over each vertex and assigns values to the matrix cells based on connectivity and weights.

Then the function returns the generated adjacency matrix.

Kruskal's Input Main Function

C/C++

```
int main() {
    int numVertices;
    std::cout << "Enter the number of vertices: ";
    std::cin >> numVertices;

    // Generating the vertex names
    std::vector<std::string> vertexNames;
    for (int i = 0; i < numVertices; i++) {
        std::string name = generateVertexName(i);
        vertexNames.push_back(name);
    }

    // Generating the adjacency matrix
    std::vector<std::vector<std::string>> adjacencyMatrix =
generateAdjacencyMatrix(numVertices);

    // Output of the output into a file.
    std::ofstream outputFile("kruskalwithoutpq_kruskalwithpq_am_" +
std::to_string(numVertices) + "_input.txt"); //converting numVertices to
string

    if (outputFile.is_open()) {
        // Output of the number of vertices
        outputFile << numVertices << std::endl;

        // Output of vertex index and name.
        for (int i = 0; i < numVertices; i++) {
            outputFile << i << " " << vertexNames[i] << std::endl;
        }

        // Output of adjacency matrix
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
```



```

        outputFile << adjacencyMatrix[i][j] << " ";
    }
    outputFile << std::endl;
}

outputFile.close();
std::cout << "File Generated." << std::endl;
}
else {
    std::cerr << "Failed to open file." << std::endl;
    return 1;
}

return 0;
}

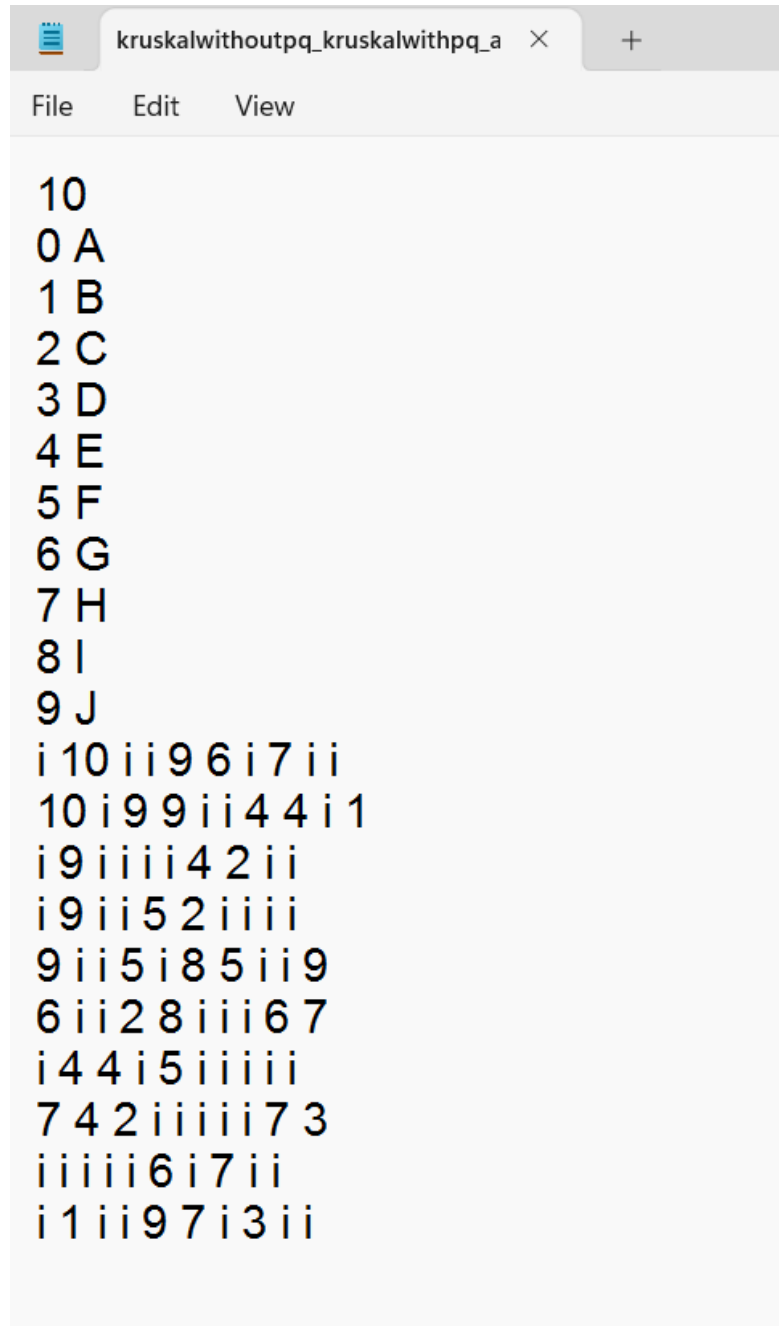
```

❖ Main Flow Function:

- Ask the user to enter the number of vertices.
- Generate vertex names based on the number of vertices.
- Generate an adjacency matrix using random connections and weights.
- Create an output file for storing the generated input.
- If the file is successfully opened:
 - Write the number of vertices to the file.
 - Write the vertex index and name pairs to the file.
 - Write the adjacency matrix to the file.
 - Close the file.
 - Display a success message.
- If the file fails to open, display an error message.

Output Analysis

Screenshot Sample



The screenshot shows a text editor window with a single tab titled "kruskalwithoutpq_kruskalwithpq_a". The window has a menu bar with "File", "Edit", and "View". The text content is as follows:

```
10
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 I
9 J
i 10 i i 9 6 i 7 i i
10 i 9 9 i i 4 4 i 1
i 9 i i i i 4 2 i i
i 9 i i 5 2 i i i i
9 i i 5 i 8 5 i i 9
6 i i 2 8 i i i 6 7
i 4 4 i 5 i i i i i
7 4 2 i i i i i 7 3
i i i i i 6 i 7 i i
i 1 i i 9 7 i 3 i i
```

Kruskal's AlgorithmOutput (Without PQ)

Code Structure & Explanation

Libraries & Usages

```
C/C++  
#include <iostream>  
#include <fstream>  
#include <vector>  
#include <algorithm>  
#include <chrono>  
#include <string>  
#include <iomanip>
```

- ❖ **<vector>**: it is used to store vertex names, edges, and the parent array for disjoint sets.
- ❖ **<algorithm>**: Used for sorting operations. It is specifically used to sort the edges based on their weights before applying Kruskal's algorithm to generate the Minimum Spanning Tree (MST).
- ❖ **<chrono>**: Used for measuring time durations. It is used to calculate the execution time of the program, which is then written to the output file.
- ❖ **<string>**: Used for string manipulation. It is used to store and process vertex names and weights read from the input file.
- ❖ **<iomanip>**: Used for formatting output. It is used to set the decimal place of the time taken, which is then written to the output file.

Edge Structure

```
C/C++
struct Edge {
    int src, dest, weight;
};
```

- ❖ The struct Edge represents an edge in the graph. It has three data members: weight, src, and dest.
 - The weight variable represents the weight (or cost) of the edge.
 - The src variable represents the source vertex of the edge.
 - The dest variable represents the destination vertex of the edge.

Finding Ultimate Parent

```
C/C++
int findParent(std::vector<int>& parent, int vertex) {
    if (parent[vertex] == vertex)
        return vertex;
    return findParent(parent, parent[vertex]);
}
```

The 'findParent' function is used to find the ultimate parent of a vertex in a disjoint set. It recursively follows the parent pointers until it reaches the root, which represents the parent of the set the vertex belongs to.

Union Sets

```
C/C++
void unionSets(std::vector<int>& parent, int x, int y) {
    int xParent = findParent(parent, x);
    int yParent = findParent(parent, y);
    parent[yParent] = xParent;
}
```

The function merges two sets together by updating their parent elements. It takes a vector called parent and two integers x and y as input. It finds the parent elements of sets x and y using the [findParent](#) function, and then sets the parent of y to be the same as the parent of

x. This process combines the two sets. The function is used to connect vertices from different sets when adding edges to the minimum spanning tree.

Compare Edges

```
C/C++
bool compareEdges(const Edge& edge1, const Edge& edge2) {
    return edge1.weight < edge2.weight;
}
```

The function compares two edges based on their weights. It returns true if the weight of the first edge is less than the weight of the second edge. It is used to sort the edges in ascending order of weight for the algorithm to select edges with smaller weights first.

Finding MST Using Kruskal's Algorithm

```
C/C++
// MST using Kruskal's algorithm.
std::vector<Edge> generateMST(int numVertices, std::vector<Edge>& edges) {
    // Sorting edges based on weight.
    std::sort(edges.begin(), edges.end(), compareEdges);

    std::vector<Edge> mst; // MST
    std::vector<int> parent(numVertices); // Parent array for disjoint sets

    // Initialize parent array
    for (int i = 0; i < numVertices; i++)
        parent[i] = i;

    int edgeCount = 0; // Number of edges in MST

    // Going through the sorted edges.
    for (const auto& edge : edges) {
        int srcParent = findParent(parent, edge.src);
        int destParent = findParent(parent, edge.dest);

        // Checking if the edge is in a cycle.
        if (srcParent != destParent) {
            mst.push_back(edge); // Add the edge to the MST
            unionSets(parent, srcParent, destParent);
            edgeCount++;
        }
    }
}
```

```

        // Stopping the algorithm if MST contains V-1 edges (V is the
        number of vertices).
        if (edgeCount == numVertices - 1)
            break;
    }
}

return mst;
}

```

This function implements Kruskal's algorithm to find the MST of a graph. It sorts the edges based on their weights, creates an empty MST vector, and initialises a parent array for disjoint sets. It iterates through the sorted edges, adding each edge to the MST if it does not create a cycle and updating the parent array accordingly. The algorithm stops when the MST contains $V-1$ edges, where V is the number of vertices. It then returns the MST vector.

Kruskal's Output (Without PQ) Main Function

```

C/C++

int main() {
    int outputVertexNum = 0;
    std::cout << "(For File Reading Purposes) \n";
    std::cout << "Number of vertcies? ";
    std::cin >> outputVertexNum;

    auto start = std::chrono::high_resolution_clock::now();
    int totalweight = 0;
    int numVertices;

    std::ifstream inputFile("kruskalwithoutpq_kruskalwithpq_am_" +
        std::to_string(outputVertexNum) + "_input.txt");

    if (inputFile.is_open()) {
        inputFile >> numVertices; // Read the number of vertices

        std::vector<std::string> vertexNames(numVertices);
        std::vector<Edge> edges;

        // Read vertex index and name
        for (int i = 0; i < numVertices; i++) {
            int index;

```

```

        std::string name;
        inputFile >> index >> name;
        vertexNames[index] = name;
    }

    // Read the adjacency matrix
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            std::string weight;
            inputFile >> weight;
            if (weight != "i") {
                int w = std::stoi(weight);
                edges.push_back({ i, j, w });
            }
        }
    }

    inputFile.close();

    std::vector<Edge> mst = generateMST(numVertices, edges);

    std::ofstream outputFile("kruskalwithoutpq_am_" +
std::to_string(numVertices) + "_output.txt");
    if (outputFile.is_open()) {
        // Write number of vertices
        outputFile << numVertices << "\n\n";

        // Write vertex index and vertex name
        for (int i = 0; i < numVertices; i++) {
            outputFile << i << " " << vertexNames[i] << "\n";
        }
        outputFile << "\n";

        // Write edge vertex pairs with edge weight
        for (const auto& edge : mst) {
            std::string srcName = vertexNames[edge.src];
            std::string destName = vertexNames[edge.dest];
            outputFile << srcName << " " << destName << " " <<
edge.weight << "\n";
            totalweight = edge.weight + totalweight;
        }

        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> duration = end - start;
        outputFile << "\n";
        outputFile << totalweight << "\n\n";
    }

```

```

        outputFile << std::fixed << std::setprecision(5) <<
duration.count() << "s" << std::endl;

        outputFile.close();
    }
    else {
        std::cerr << "Failed to create the output file." << std::endl;
        return 1;
    }

}
else {
    std::cerr << "Failed to open the input file." << std::endl;
    return 1;
}

return 0;
}

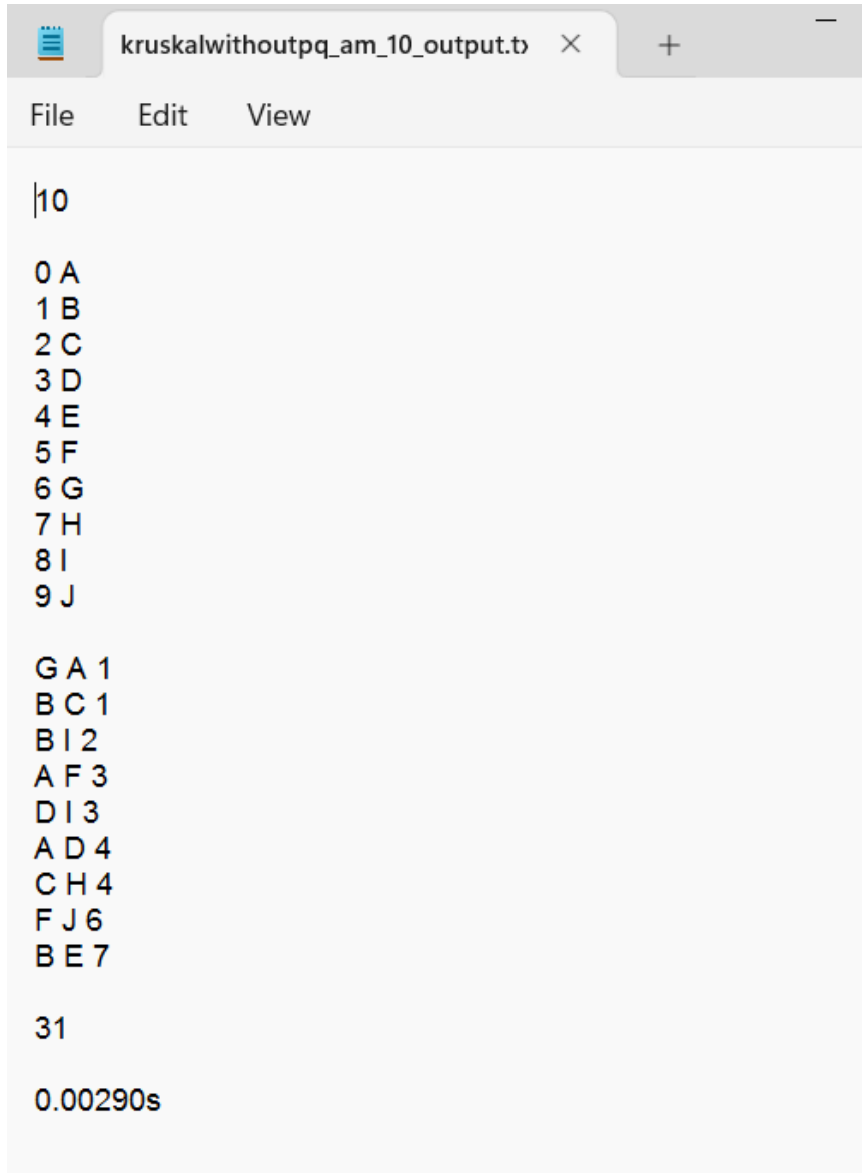
```

Main Function Flow:

- ❖ Read the number of vertices from the user.
- ❖ Start measuring the execution time.
- ❖ Initialise variables for total weight and number of vertices.
- ❖ Open the input file for reading.
- ❖ If the file is open:
 - Read the number of vertices from the file.
 - Create vectors for vertex names and edges.
 - Read vertex index and name from the file.
 - Read the adjacency matrix and create edges for non-"i" weights.
 - Close the input file.
 - Generate the Minimum Spanning Tree (MST) using generateMST function.
 - Open the output file for writing.
 - If the file is open:
 - Write the number of vertices to the file.
 - Write vertex index and vertex name to the file.
 - Write edge vertex pairs with edge weight to the file and calculate the total weight.
 - Measure the elapsed time and write it to the file.
 - Close the output file.
 - If the output file could not be created, display an error message.
- ❖ If the input file could not be opened, display an error message.

Output Analysis

Screenshot Sample



A screenshot of a text editor window titled "kruskalwithoutpq_am_10_output.txt". The window has a menu bar with "File", "Edit", and "View". The text content is as follows:

```
|10  
  
0 A  
1 B  
2 C  
3 D  
4 E  
5 F  
6 G  
7 H  
8 I  
9 J  
  
GA 1  
BC 1  
BI 2  
AF 3  
DI 3  
AD 4  
CH 4  
FJ 6  
BE 7  
  
31  
  
0.00290s
```

Kruskal's AlgorithmOutput (With PQ)

Code Structure & Explanation

Libraries & Usages

```
C/C++  
#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
#include <queue>  
#include <algorithm>  
#include <chrono>  
#include <iomanip>
```

- ❖ **<vector>**: Used to store dynamic data structures like vertex names, adjacency matrix, and other resizable collections.
- ❖ **<queue>**: Used to implement a priority queue for Kruskal's algorithm, allowing constant time retrieval of the smallest element.
- ❖ **<algorithm>**: Used for sorting the edges in the priority queue.
- ❖ **<chrono>**: Used for measuring time durations. It is used to calculate the execution time of the program.
- ❖ **<iomanip>**: Used for formatting output. It is used to set the decimal place of the time taken, which is then written to the output file.

Edge Structure

```
C/C++
struct Edge {
    int weight, u, v;

    Edge(int weight, int u, int v) : weight(weight), u(u), v(v) {}
};
```

- ❖ The struct Edge represents an edge in the graph. It has three data members: weight, src, and dest.
 - The weight variable represents the weight (or cost) of the edge.
 - The src variable represents the source vertex of the edge.
 - The dest variable represents the destination vertex of the edge.
- ❖ The constructor of the Edge struct initialises these data members with the given values.

Compare Edges

```
C/C++
struct CompareEdges {
    bool operator()(const Edge& a, const Edge& b) const {
        return a.weight > b.weight;
    }
};
```

The function compares two edges based on their weights. It returns true if the weight of the first edge is less than the weight of the second edge. It is used to sort the edges in ascending order of weight for the algorithm to select edges with smaller weights first.

Find & Unite Sets

```
C/C++
class UnionFind {
private:
    std::vector<int> parent;
    std::vector<int> rank;

public:
    UnionFind(int size) {
        parent.resize(size);
        rank.resize(size, 0);

        for (int i = 0; i < size; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {
        if (x != parent[x]) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return;
        }

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        }
        else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        }
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
};
```

The 'UnionFind' class implements the Union-Find data structure, which has 'find' and 'unite' operations to find the representative and unite sets. It uses vectors for parent and rank. 'find' applies path compression, and 'unite' combines sets based on rank.

The '[kruskalMST](#)' function uses Union-Find to implement Kruskal's algorithm for finding the MST. It uses a priority queue to sort edges by weight. It checks for cycles and adds edges to the MST.

In 'main', input is read from a file, and the MST is computed using 'kruskalMST'. The output is written to a file, including the MST and total weight.

Finding MST Using Kruskal's Algorithm

```
C/C++
void kruskalMST(const std::vector<std::vector<std::string>>&
adjacencyMatrix, const std::vector<std::string>& vertexNames, std::ofstream&
outputFile) {
    int numVertices = vertexNames.size();
    std::priority_queue<Edge, std::vector<Edge>, CompareEdges> pq;

    // Add all edges to the priority queue
    for (int i = 0; i < numVertices; i++) {
        for (int j = i + 1; j < numVertices; j++) {
            if (adjacencyMatrix[i][j] != "i") {
                int weight = std::stoi(adjacencyMatrix[i][j]);
                pq.push(Edge(weight, i, j));
            }
        }
    }

    UnionFind uf(numVertices);
    std::vector<Edge> mst;
    int totalWeight = 0;

    // Kruskals Algo.
    while (!pq.empty()) {
        Edge currentEdge = pq.top();
        pq.pop();

        if (uf.find(currentEdge.src) != uf.find(currentEdge.dest)) {
            uf.unite(currentEdge.src, currentEdge.dest);
            mst.push_back(currentEdge);
            totalWeight += currentEdge.weight;
        }
    }
}
```

```

// Output the results to the file
outputFile << numVertices << std::endl;

outputFile << std::endl;
for (int i = 0; i < numVertices; i++) {
    outputFile << i << " " << vertexNames[i] << std::endl;
}

outputFile << std::endl;
for (const Edge& edge : mst) {
    outputFile << vertexNames[edge.src] << " " << vertexNames[edge.dest]
<< " " << edge.weight << std::endl;
}
outputFile << "\n";
outputFile << totalWeight << "\n\n";
}

```

The 'kruskalMST' function implements Kruskal's algorithm to find the minimum spanning tree (MST) of a graph. It takes an adjacency matrix, vertex names, and an output file stream as input.

It initialises variables and a priority queue for storing edges. Then it adds valid edges to the priority queue.

In a loop, it selects edges with the minimum weight and checks for cycles. If an edge doesn't create a cycle, it adds it to the MST and updates the total weight.

Then it writes the results to the output file, including the number of vertices, vertex names, and the MST edges with their weights.

Kruskal's Output (With PQ) Main Function

```

C/C++

int main() {
    int outputVertexNum = 0;
    std::cout << "(For File Reading Purposes) \n";
    std::cout << "Number of vertcies? ";
    std::cin >> outputVertexNum;
    auto start = std::chrono::high_resolution_clock::now();
    std::ifstream inputFile("kruskalwithoutpq_kruskalwithpq_am_" +
std::to_string(outputVertexNum) + "_input.txt");
    if (!inputFile.is_open()) {
        std::cerr << "Failed to open the input file." << std::endl;
        return 1;
    }
}

```

```

    }

    int numVertices;
    inputFile >> numVertices;

    std::vector<std::string> vertexNames(numVertices);
    for (int i = 0; i < numVertices; i++) {
        int index;
        std::string name;
        inputFile >> index >> name;
        vertexNames[index] = name;
    }

    std::vector<std::vector<std::string>> adjacencyMatrix(numVertices,
std::vector<std::string>(numVertices));
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            inputFile >> adjacencyMatrix[i][j];
        }
    }

    inputFile.close();
    /*std::ofstream outputFile("kruskalwithoutpq_kruskalwithpq_am_" +
std::to_string(numVertices) + "_input.txt");*/
    std::ofstream outputFile("kruskalwithpq_am_"
+std::to_string(numVertices) + "_output.txt");
    if (!outputFile.is_open()) {
        std::cerr << "Failed to open the output file." << std::endl;
        return 1;
    }

    // Time taken:

    kruskalMST(adjacencyMatrix, vertexNames, outputFile);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    outputFile << std::fixed << std::setprecision(5) << duration.count() <<
" seconds" << std::endl;

    outputFile.close();

    std::cout << "Compelte!";

    return 0;
}

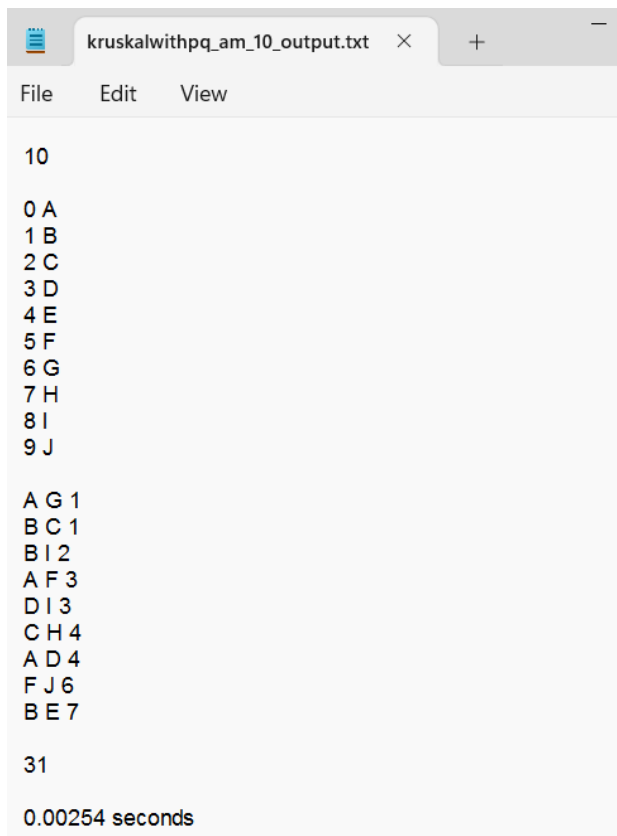
```

❖ Main Function Flow:

- Prompt the user for the number of vertices.
- Read the input file with the specified number of vertices.
- If the input file fails to open, display an error message and exit.
- Read the number of vertices from the input file.
- Create a vector 'vertexNames' to store the names of vertices.
- Read the vertex indices and names from the input file and store them in 'vertexNames'.
- Create a 2D vector 'adjacencyMatrix' to store the adjacency matrix of the graph.
- Read the adjacency matrix from the input file and store it in 'adjacencyMatrix'.
- Close the input file.
- Open the output file for writing the MST results.
- If the output file fails to open, display an error message and exit.
- Start measuring the execution time.
- Call the '[kruskalMST](#)' function with the adjacency matrix, vertex names, and output file.
- Stop measuring the execution time.
- Write the duration of the execution to the output file.
- Close the output file.
- Display "Complete!" message.

Output Analysis

Screenshot Sample



```
10

0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
8 I
9 J

A G 1
B C 1
B I 2
A F 3
D I 3
C H 4
A D 4
F J 6
B E 7

31

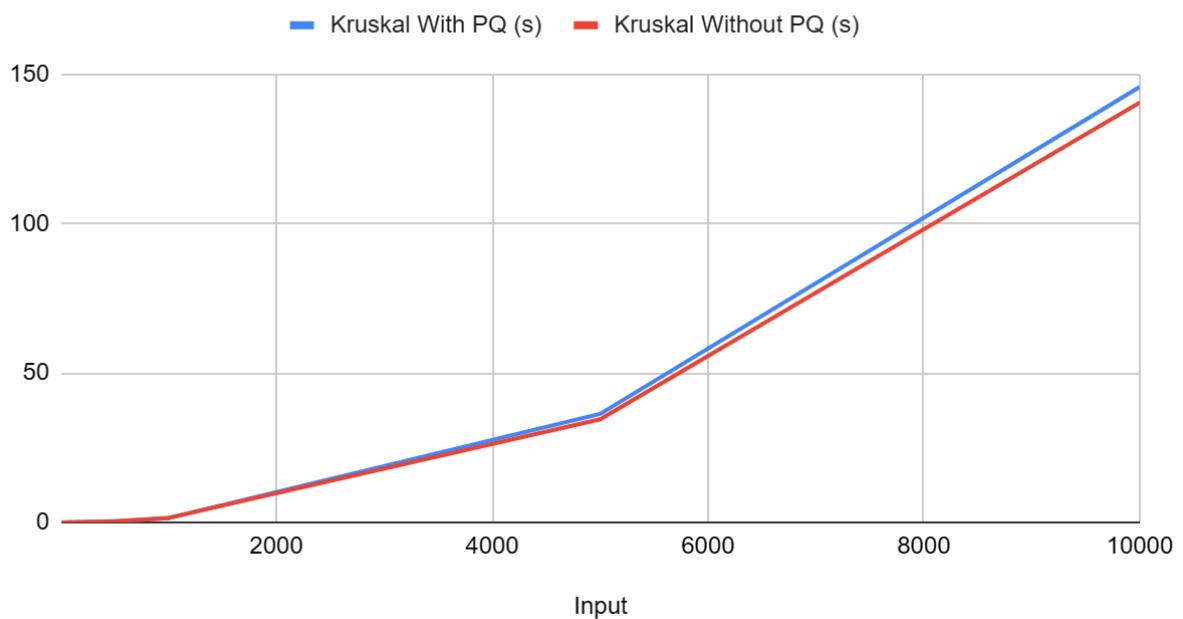
0.00254 seconds
```


Kruskal's - Comparative Analysis

Input Size VS Time

Input	Kruskal With PQ (s)	Kruskal Without PQ (s)
10	0.0074	0.00125
100	0.0219	0.01606
500	0.35866	0.34207
1000	1.4	1.4438
5000	36.31056	34.4975
10000	145.715	140.50288

Kruskal With PQ (s) and Kruskal Without PQ (s)



Theoretically, Kruskal's algorithm with PQ is supposed to be more efficient than without a PQ.

Although, Some things do have to be taken into consideration:

The Kruskal's Algorithm Without PQ i have written still uses `sort()` to sort the edges. Knowing this, it can be concluded that the algorithm without PQ is still faster, even though it uses `sort()` instead of PQ.

Conclusion

In conclusion, the observed difference in efficiency between Kruskal's algorithm without a priority queue (PQ) and Kruskal's algorithm with a PQ can be attributed to what type of data structures were used, and the overall implementation of the program. The first implementation utilises a vector and sorting, resulting in a time complexity of $O(E \log E)$, while the second implementation uses a PQ for efficient edge retrieval, resulting in a time complexity of $O(E \log V)$.

Both algorithms have the same inputs and outputs, so the I/O cannot be the reason for the slight difference in efficiency.

Both algorithms were run on the same hardware as well, so hardware difference is not the reason for the efficiency differences.

The time complexity of the implementation of UnionFind is about the same too. Therefore, the reason for the difference in efficiency may be attributed to the data structures used & the libraries used. I have used the <algorithm> library in the algorithm with the PQ. which may have contributed to the difference.

The overall reason for the difference is inconclusive. But, more towards the type of data structures used & the libraries used.

The overall conclusion for the efficiency of the algorithm is that the algorithm without PQ is slightly faster than the algorithm with the PQ, although they have the same time complexity.

Appendix

Reference

[Graph Series by Striver | C++ | Java | Interview Centric | Algorithms | Problems - YouTube](#)

[\(27\) G-46. Disjoint Set | Union by Rank | Union by Size | Path Compression - YouTube](#)

[\(27\) G-47. Kruskal's Algorithm - Minimum Spanning Tree - C++ and Java - YouTube](#)

[HUFFMAN ENCODING AND DECODING IMPLEMENTATION C++ | UNEDITED | FULL ILLUSTRATION | BORING WARNING !!! - YouTube](#)

[Huffman Implementation - YouTube](#)

Assistance

1. My Elder Brother.
 - a. Guided me throughout the project by showing me the general idea of each algorithm and their explanations.
2. Chat GPT
 - a. Debugging code, guidance & explanation of how the algorithms may be implemented.
 - i. Although Chat GPT was used for this project, no code was blindly copied.

Bugs & Fixes

Huffman Input

There is a bug where the number of unique characters are not actually used to determine how many unique characters the program will use to generate the words. I did not have enough time to fix this bug, as Huffman was the last algorithm I wrote. But, I doubt it would be hard.

A way to go about this bug would be to create a randomly generated string that only contains the unique characters, instead of creating a string of all the characters in the alphabet.

My plan was to use that string and then pick a few characters from that string, but it did not work out. So, I ended up leaving it as it was.

Kruskal's Algorithm

Generating the input file for 100,000 characters took a very, very long time. I stopped the program while it was finishing up with the input, and the size of the file was about 7GBs.

I decided to try to improve the algorithm by using Multi Threading.

The attempt was successful for the smaller inputs. But, when an attempt was made to write the input file using multi-threading, I got a device warning, which I had never gotten before. It was not even from the compiler, but from my operating system. I then decided to ditch the idea of using multi threading.

But, if multithreading was used properly, the efficiency for generating the input file or even generating the output file would be drastically better. This is because the input is writing a matrix. The matrix could be divided into lines and the threads could then run concurrently to handle each line separately and then joined to produce the matrix much more efficiently.

And the output for 100,000 could not be generated. After a few hours of running the code, there was an abort message.