MULTIMEDIA UNIVERSITY

FACULTY OF COMPUTING INFORMATION

TCP2101 ALGORITHM DESIGN AND ANALYSIS

Trimester 2, 2022/2023

Lecture section: TC2L
Group number: 205
Group leader student name: Chia Yu Zhang

Num	Student ID	Student Name	Task Descriptions	Percentage %
1	1201101003	CHIA YU ZHANG	Q5,Q6, Q7,Q8,Q9,Q10	25
2	1201100324	GERALD GODWIN LEE YONG LIN	Q1,Q2,Q3,Q4,Q8,Q9,Q10	25
3	1201100924	TEO HAZEL	Q1,Q2,Q3,Q4,Q8,Q9,Q10	25
4	1201100514	YAP ZHI TOUNG	Q5, Q6,Q7, Q8,Q9,Q10	25
			Total:	100%

Mark sheet checklist to be filled by students at certain parts (40%)

No.	Task	Mark	Checklist (Yes/No) with explanation if any and write only one student name to be responsible for each task number, must be filled by students
1	Kruskal algorithm without priority queue and adjacency matrix graph of n number of vertices implementation for file inputs and file outputs with screen outputs with step-by-step illustration for the minimum spanning tree problem where n=6. input filename: kruskalwithoutpq_am_0000006_input.txt output filename: kruskalwithoutpq_am_0000006_output.txt	5	Yes(Teo Hazel)
2	Adjacency matrix complete graphs for Kruskal algorithm of n number of vertices implementation for dataset generation of input files that contain random edge weight integers in each file (10 vertices, 100 vertices, 1000 vertices, 10000 vertices, 100000 vertices, etc.). Write a function to generate all the input files. The filenames are: - kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt - kruskalwithoutpq_kruskalwithpq_am_00000100_input.txt - kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt - kruskalwithoutpq_kruskalwithpq_am_00010000_input.txt - kruskalwithoutpq_kruskalwithpq_am_00100000_input.txt	5	Yes(Teo Hazel)
3 4	Kruskal algorithm without priority queue and with priority queue of n number of vertices for input files of different problem sizes that have been generated previously and output files with screen outputs with algorithm times for the minimum spanning tree problem. Write a function to generate all output files for each input size n. The filenames without priority queue are: - kruskalwithoutpq_am_00000010_output.txt	10	(Yes)Gerald Godwin Lee Yong Lin

	- kruskalwithoutpq_am_00000100_output.txt - kruskalwithoutpq_am_00001000_output.txt - kruskalwithoutpq_am_00010000_output.txt - kruskalwithoutpq_am_00100000_output.txt The filenames with priority queue are: - kruskalwithpq_am_00000010_output.txt - kruskalwithpq_am_00000100_output.txt - kruskalwithpq_am_00001000_output.txt - kruskalwithpq_am_00010000_output.txt - kruskalwithpq_am_00100000_output.txt		
5	Huffman coding of n number of words implementation for file inputs and file outputs with screen outputs with step-by-step illustration for the lossless data compression problem where n=3. input filename: huffmancoding_00000003_input.txt output filename: huffmancoding_00000003_output.txt	5	(YES)YAP ZHI TOUNG
6	Random words for Huffman coding algorithm of n number of words implementation for dataset generation of input files that contain random words in each file (10 words, 100 words, 1000 words, 10000 words, 100000 words, etc.). Write a function to generate all the input files. The filenames are: - huffmancoding_00000010_input.txt - huffmancoding_00000100_input.txt - huffmancoding_00001000_input.txt - huffmancoding_00010000_input.txt - huffmancoding_00100000_input.txt	5	(YES)CHIA YU ZHANG
7	Huffman coding algorithm of n number of words for input files of different problem sizes that have been generated previously and output files with screen outputs with algorithm space percentages. The filenames are: - huffmancoding_00000010_output.txt - huffmancoding_00000100_output.txt - huffmancoding_00001000_output.txt - huffmancoding_00010000_output.txt - huffmancoding_00100000_output.txt	5	(YES)YAP ZHI TOUNG

8 9	Your report of screenshots, code parts, explanations, input files, output files and step-by-step illustration contains the following. * Kruskal algorithm without priority queue and Kruskal algorithm with priority queue using adjacency matrix complete graph for the minimum spanning tree ** perform numerous experiments of different input sizes using the algorithms, get the total times for the algorithms ** the above experiment results that can be used to perform a comparative analysis between the two implementations in table form and graph form ** conclude your findings in the report * Lossless data compression using Huffman coding algorithm ** perform numerous experiments of different input sizes using the algorithm, get the total space percentages for the algorithm ** the above experiment results that can be used to perform a comparative analysis between with data compression and without data compression in table form and graph form ** conclude your findings in the report	10	(YES)CHIA YU ZHANG
10	Group video presentation with faces with a maximum of twenty minutes. Make an appointment for your group interview and meeting with your tutor to validate your work.	5	(YES) CHIA YU ZHANG, GERALD GODWIN LEE YONG LIN, TEO HAZEL, YAP ZHI TOUNG
	Assignment mark	50	

TABLE OF CONTENT

Algorithm 1:Kruskal algorithm without priority queue

Question 1 :

Kruskal algorithm without priority queue and adjacency matrix graph of n number of vertices implementation for file inputs and file outputs with screen outputs with step-by-step illustration for the minimum spanning tree problem.

Input filename : huffmancoding_00000003_input.txt	description	Output filename: huffmancoding_00000003_input.txt	description

6
0 A
1 B
2 C
3 D
4 E
5 F
i 2 8 i 7 i
2 i 5 7 i i
8 5 i 9 8 i
i 7 9 i i 4
7 i 8 i i 3
i i i 4 3 i
	// num of vertices
	// vertices indexes andvertex names
	
	// adjacency matrix
		6
		0 A
		1 B
		2 C
		3 D
		4 E
		5 F
		A B 2
		E F 3
		D F 4
		B C 5
		A E 7
		21
		10s
			// num of vertices
			// vertices indexes andvertex names
			
			//edge vertex pairs,edge weights in alphabetical order
			
			//total weight
			//example total time taken

Code for Output File

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    vector<Edge> edges;
};

bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

int findParent(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return findParent(parent, parent[i]);
}

void unionSet(int parent[], int x, int y) {
    int xset = findParent(parent, x);
    int yset = findParent(parent, y);
    parent[xset] = yset;
}

void kruskalMST(Graph& graph, ofstream& output) {
    int V = graph.V;
    vector<Edge> result;
    int* parent = new int[V];

    // Record the start time
    auto start = chrono::system_clock::now ();

    for (int i = 0; i < V; i++)
        parent[i] = i;

    sort(graph.edges.begin(), graph.edges.end(), compareEdges);

    int i = 0, e = 0;
    while (e < V - 1 && i < graph.E) {
        Edge nextEdge = graph.edges[i++];

        int x = findParent(parent, nextEdge.src);
        int y = findParent(parent, nextEdge.dest);

        if (x != y) {
            result.push_back(nextEdge);
            unionSet(parent, x, y);
            e++;
        }
    }

    // Record the end time
    auto end = chrono::system_clock::now ();

    // Calculate the duration
    chrono::duration < double >duration = end - start;

    int totalWeight = 0;
```

```
    output << V << endl;
    for (int i = 0; i < V; i++)
        output << i << " " << (char)('A' + i) << endl;

    for (Edge edge : result) {
        output << (char)('A' + edge.src) << " " << (char)('A' + edge.dest) << " " << edge.weight << endl;
        totalWeight += edge.weight;
    }

    output << totalWeight << endl;

    // Print the total time taken in second
    // cout << duration.count () << "s" << endl;
    output << duration.count () << "s" << endl;

    delete[] parent;
}

int main() {
    string inputFileName = "kruskalwithoutpq_am_0000006_input.txt";
    string outputFileName = "kruskalwithoutpq_am_0000006_output.txt";

    ifstream inputFile (inputFileName);

    if (!inputFile)
    {
        cout << "Failed to open the input file." << endl;
        return 0;
    }

    Graph graph;
    inputFile >> graph.V;

    for (int i = 0; i < graph.V; i++) {
        int index;
        char name;
        inputFile >> index >> name;
    }

    for (int i = 0; i < graph.V; i++) {
        for (int j = 0; j < graph.V; j++) {
            int weight;
            char c;
            inputFile >> c;
            if (c == 'i')
                continue;
            else
                weight = c - '0';
            if (i < j) {
                Edge edge;
                edge.src = i;
                edge.dest = j;
                edge.weight = weight;
                graph.edges.push_back(edge);
            }
        }
    }

    graph.E = graph.edges.size();

    inputFile.close();

    ofstream outputFile(outputFileName);

    if (!outputFile)
    {
        cout << "Failed to open the output file." << endl;
        return 0;
    }

    kruskalMST(graph, outputFile);

    outputFile.close();

    cout << "Output generated and saved to kruskalwithoutpq_am_0000006_output.txt." << endl;

    return 0;
}
```

Figure 1.1: Code for Question 1

```
Output generated and saved to kruskalwithoutpq_am_0000006_output.txt.

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

Figure 1.2: Screen run

Explanation

The provided code implements Kruskal’s algorithm to find the minimum spanning tree (MST) of a given graph. It reads the graph information from an input file, sorts the edge based on their weights, and then iteratively selects edges with the minimum weight that do not form cycles in the MST. The resulting MST is written to an output file, along with the total weight and the execution time of the algorithm. Kruskal’s algorithm is an efficient approach to find the minimum spanning tree of a graph, making it useful in various applications where finding an optimal connected subgraph is required.

Output File

kruskalwithoutpq_am_0000006_output.txt.

```
6
0 A
1 B
2 C
3 D
4 E
5 F
A B 2
E F 3
D F 4
B C 5
A E 7
21
4.7e-06s

```

Figure 1.3: Output file/ Answer for Question 1

Question 2 :

Adjacency matrix complete graphs for Kruskal algorithm of n number of vertices implementation for dataset generation of input files that contain random edge weight integers in each file (10 vertices, 100 vertices, 1000 vertices, 10000 vertices, 100000 vertices, etc.). Write a function to generate all the input files.

The filenames are:

- kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt
- kruskalwithoutpq_kruskalwithpq_am_00000100_input.txt
- kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt
- kruskalwithoutpq_kruskalwithpq_am_00010000_input.txt
- kruskalwithoutpq_kruskalwithpq_am_00100000_input.txt

Code for Output File

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <random>

using namespace std;

// Function to generate a random integer between min and max (inclusive)
int generateRandomInt(int min, int max) {
    static random_device rd;
    static mt19937 gen(rd());
    uniform_int_distribution<> dis(min, max);
    return dis(gen);
}

// Function to generate the input file for Kruskal algorithm with adjacency matrix representation
void generateInputFile(int numVertices, const string& filename) {
    ofstream file(filename);

    if (file.is_open()) {
        file << numVertices << "\n";

        for (int i = 0; i < numVertices; i++) {
            file << i << " " << "A" << i << "\n";
        }

        vector<vector<int>> adjacencyMatrix(numVertices, vector<int>(numVertices, 0));

        for (int i = 0; i < numVertices; i++) {
            for (int j = i + 1; j < numVertices; j++) {
                int weight = generateRandomInt(1,1201101003);
                adjacencyMatrix[i][j] = weight;
                adjacencyMatrix[j][i] = weight;
```

```cpp
                }
            }


            for (int i = 0; i < numVertices; i++) {
                for (int j = 0; j < numVertices; j++) {
                    if (i == j) {
                        file << "i";
                    } else {
                        file << adjacencyMatrix[i][j];
                        adjacencyMatrix[j][i] = adjacencyMatrix[i][j]; // Assign the same weight to adjacencyMatrix[j][i]
                    }

                    if (j != numVertices - 1) {
                        file << " ";
                    }
                }
                file << "\n";
            }

            file.close();
            cout << "Input file " << filename << " generated successfully.\n";
        } else {
            cerr << "Unable to create the file: " << filename << "\n";
        }
}

int main() {
        string filePrefix = "kruskalwithoutpq_kruskalwithpq_am_";
        string fileSuffix = "_input.txt";
        int inputSizes[] = {10, 100, 1000, 5000, 10000, 100000};
        int numInputSizes = sizeof(inputSizes) / sizeof(inputSizes[0]);

        for (int i = 0; i < numInputSizes; i++) {
            string paddedInputSizeStr = string(8 - to_string(inputSizes[i]).length(), '0') + to_string(inputSizes[i]);
            string filename = filePrefix + paddedInputSizeStr + fileSuffix;
            generateInputFile(inputSizes[i], filename);
        }

        return 0;
}
```

Figure 2.1: Code for Question 2

```
"C:\Algorithm\Kruskal algorit
Input file kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt generated successfully.
Input file kruskalwithoutpq_kruskalwithpq_am_00000100_input.txt generated successfully.
Input file kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt generated successfully.
Input file kruskalwithoutpq_kruskalwithpq_am_00005000_input.txt generated successfully.
Input file kruskalwithoutpq_kruskalwithpq_am_00010000_input.txt generated successfully.

Process returned 3 (0x3)   execution time : 48.245 s
Press any key to continue.
```

Figure 2.2: Screen run

Explanation

This code generates input files for the Kruskal algorithm with an adjacency matrix representation. The program uses a random number generator to assign weights to the edges between vertices in a graph. The generated input files can be used as input for testing and implementing the Kruskal algorithm, which finds the minimum spanning tree of a connected, undirected graph. The code loops over different graph sizes, creates output files with specific naming conventions, and writes the graph information to the files. The adjacency matrix representation allows for easy manipulation and analysis of the graph structure. Overall, this code automates the process of generating input files for the Kruskal algorithm with adjacency matrix representation, facilitating the testing and evaluation of the algorithm's performance.

Output File

kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt

```
10
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
i 979114877 1177551288 786554198 936563419 137257722 1013523584 36882129 415416430 461917362
979114877 i 780329188 432162013 879569557 756234313 751764663 799212319 34782306 355028890
1177551288 780329188 i 556046821 871981410 909267571 1140442023 940604377 712113355 979510324
786554198 432162013 556046821 i 220804593 351427404 1058345193 1046567301 1123304715 1055562000
936563419 879569557 871981410 220804593 i 135988061 668089062 1083406764 597480552 655678588
137257722 756234313 909267571 351427404 135988061 i 533657275 63297687 220302984 189254917
1013523584 751764663 1140442023 1058345193 668089062 533657275 i 31247930 149332081 793046364
36882129 799212319 940604377 1046567301 1083406764 63297687 31247930 i 689074503 1004773850
415416430 34782306 712113355 1123304715 597480552 220302984 149332081 689074503 i 389770443
461917362 355028890 979510324 1055562000 655678588 189254917 793046364 1004773850 389770443 i

```

Figure 2.3: Output file for “kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt”

kruskalwithoutpq_kruskalwithpq_am00000100_input.txt

```
100
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
834150600 351260490 613011859 98544198 100415818 886224978 1145180054 384446299 409598134 21764010 309700256 438086976 890743376 633410150 692650346 64970822
632051656 437439671 1035618613 939344887 1151909647 163611136 464663707 931247430 920522155 326436502 918744734 684000835 16903954 209695888 419695125 106014
110126885 632059344 49445888 108599233 125411938 518790158 912232762 328765940 296623731 419964509 268313960 769754548 505483797 880637255 761662319 95794357
306015371 754380172 258087946 58692585 58946222 952906459 883205164 294346294 506324389 223376757 44278629 774277177 658962599 702929647 556899307 756503517
842620574 896479058 895568688 846590178 174071477 960037011 847066717 593894675 685747538 536747458 742085281 1008178690 56487376 931097542 1065540619 595809
572399872 465296643 1013527334 261813115 571522286 1042034393 178055938 30981044 628439681 1170687037 759847448 1113689805 496394572 1122657271 42150570 5904
60230854 136537903 220309899 880702825 569547558 1180918288 281436007 548342695 441204100 1184935577 1153028765 1134255734 627880076 851811860 553444968 1008
1120758722 1063759752 386452456 632876378 122706666 1106871651 990416004 372139023 475945116 131967981 172679894 744338165 460026524 451431660 33007184 95051
947991296 130990017 1039754835 640790651 1138627400 777897731 452441408 537381095 968135400 1189177166 807397030 88768284 850715767 546052263 929706051 11694
626772812 919738772 361308059 372686174 570214883 961006274 1187340418 276495811 92450509 535956064 453062051 294463682 869466182 256895724 868200032 5834091
780122036 914983703 487647153 267660694 840654911 816890939 1010126046 651390820 209611318 22777698 868763993 718793416 488772167 533977795 1004752749 108535
147382279 1112076988 198054421 790302738 787936243 171291843 78739235 679522854 881078773 951743831 1175172477 1125976051 839183414 395277137 889183929 10666
1032341358 767681688 970030005 206929146 768859073 609838494 553872082 32110075 1082918722 728683820 178455609 1078849633 1969961 1034172677 1170316619 19541
106252604 83018259 142516951 817131677 33083099 380404539 612129613 590183475 507067733 945373028 54882505 98883546 506843092 481191063 565507817 598615746 3
13348651 510430558 49856188 676555907 622755797 412241667 453382298 112153725 1027334291 239081793 802508847 901366805 259837019 973184307 747139031 57565922
393662517 928974464 86482570 827149203 177332192 245262308 460679357 744092236 57418144 726504948 221711560 442990019 65732774 468153961 1026363485 167569450
906865135 807161758 27758543 388818387 540857140 869550703 96277852 461515096 603905541 55027441 622018009 181456399 212231247 581424370 957305691 926880155
761613179 1052257356 634266582 214337791 705928659 915687798 987635041 1158756554 287209041 265561798 43761646 802045853 150332824 15567384 754565970 3924936
593144287 996991057 524507076 67889497 116344613 442765773 947677070 120924881 792438886 683202619 1098835436 1096641405 1060925385 476382100 1166129299 2480
191912388 25159352 989986907 39588300 475419636 338166650 842112401 1008610616 818747109 72007383 7347798 773902248 952573052 280484066 987690159 262117775 4
415052250 909123458 713484556 1129433425 743455552 760662087 72995701 292415582 1052897803 414677337 287115609 899365047 1076455100 982558335 343793018 11625
266550587 933780249 798072870 552433506 457073575 515745316 840226823 1122785620 726729604 838136191 589841593 149560370 1074646674 88781781 227879955 981034
954451872 219216087 1034400196 844538490 643663170 755682300 557137074 290840791 647916417 366021175 782543404 1109929161 586296740 109789323 81250676 270080
353065400 541680277 276425251 871564122 267723478 114193679 423761274 96135636 125561650 155420521 1120560847 298750956 205487920 1172946822 91700680 5524228
1046386604 867968546 811690996 1001739511 387392274 49076912 528840976 513462660 106153838 857418907 423904802 469014003 127605960 860878705 166518229 128069
481656370 1102843285 498446202 140090395 684732311 647624748 5170859 616752690 771567588 249401531 796230403 462328953 321733126 485100307 540262294 75186153
801657164 559570463 228802588 770923430 729488831 1088886336 163659049 336301265 691257433 875633158 206881919 1173670820 220660742 695179828 724576354 24616
381851126 795496881 989834969 78807810 1041812825 105886344 960101371 783415524 194669552 1163321655 349314481 147642005 247495207 893085004 442046082 209916
348265714 78768888 273383675 659689673 551897445 303805278 268564791 396304817 97961966 499783046 817544823 293598331 472110939 392231665 1171268711 26354504
755254959 935779212 846516522 1073255052 326735660 68963204 1034607512 985616222 649734970 186745366 514018141 252135986 802715526 959343258 692205412 111285
182478415 920797072 906425408 1141393974 95048920 61908444 667982618 391464449 396767461 522565433 1103573556 906769287 520302144 749407182 1177337726 348824
1189567438 726615716 20585123 997301538 1095739315 629366827 3120247 402429810 701349531 1090369761 517914156 339718319 258279825 102317527 292342322 7502722
```

Figure 2.4: Output file for “kruskalwithoutpq_kruskalwithpq_am_00000100_input.txt”

kruskalwithoutpq_kruskalwithpq_am00001000_input.txt

```
1000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
770895335 1170825967 34542413 98332783 511577660 140104384 755606957 701297725 412316778 863501675 1118861933 932534808 234036884 212476206 540946710 2526238
930481405 476689551 1186887708 480193865 1035011478 526033100 663156861 214462684 65715763 411012523 31287808 1163508661 1113742119 69173359 1060349893 68723
12126891 845293467 741391803 643551920 801349751 551370612 179441607 311257624 918427673 1006203344 379660944 634744028 65218878 897454498 680758299 59138685
49099743 1083787440 736387526 181700896 14229170 386055791 535999377 797960631 846516318 488830864 285165977 801962689 705699417 193597153 212984984 75671442
154959950 746995959 657288901 1038813506 201716632 668151005 526819624 753814353 903553375 308833365 466013533 394964101 870274595 746356312 1071633107 89904
705513272 391298119 276985219 873032507 409001382 659153152 709912977 248860158 84191656 751904635 384623521 959404853 905854184 775653007 339463421 94624757
263383915 613674766 1099560675 753998886 1159885476 1133848267 582135361 75421915 91264777 840941995 740458381 396182257 68953092 112105349 951653483 3973780
16756381 93073825 6565354 561214529 818008334 278105763 720842162 1069217468 1137409847 644865769 800193654 374192959 171211453 934495703 192107467 747012104
946502008 939395258 675188933 861858814 1013670910 971845415 648850757 1131506192 151786225 806729119 911535606 376007178 610344442 831003048 835122406 78396
179416174 261295924 365681476 510930132 75232622 132188429 930160979 329403952 1009070821 526226155 1134207697 1021900842 17047844 201682051 940133266 116144
17326549 981810304 81431081 890469843 817869704 604303022 579506694 79602560 192813593 425272992 1181132687 50507577 456819432 364226438 224544589 251801486
147072720 939233948 934813668 841262621 611468773 44776542 876839304 465764447 638834319 125849024 715171012 50986687 1033283182 511413519 205284303 34238927
74844012 539635244 693328449 581503023 673537143 1149628273 542203922 479493277 107815533 1110742092 648659779 324330559 96813527 84380049 872021775 67994818
794636016 433279108 865185358 779915832 747142953 984697271 800451909 276550672 551020407 274946792 466525694 1146831684 585500386 172442436 663937706 998909
70349821 566106378 183073627 482500088 668774154 840166142 231050142 331324606 138680660 1033138416 78078117 524228039 984553171 1010839847 1127953721 105792
505589511 598793357 838665599 249257079 553451257 606269869 1073925717 716985547 561374785 159073554 11677004 854943013 922473845 995118698 977151024 3950342
988404816 145116003 356218856 816834314 1187629905 498770754 26794789 402181193 455104891 1037968088 248645212 892040623 211045034 912894733 862819455 276589
408247281 216769384 307621792 679777597 384796856 423787591 834127426 718431135 1004751227 860157852 388658128 809601965 845080401 703321799 491931882 572524
152498707 893624219 502185545 349551145 861723090 666239577 957799582 566823552 119126699 271629884 511987333 249238046 744903372 390908945 766894493 1564238
762211816 27856506 805425549 1120276619 928420739 989266669 718925712 747701476 682094506 351543715 327332511 746147283 337225300 1118213616 326479342 656330
338814790 468903035 761199240 43378405 991180199 778619079 839009377 40798047 926099721 219280190 909572125 943227447 1010613788 387418825 75636651 165508854
831915934 1195114005 1011129488 851560036 569111329 188802577 708360959 320963093 1138292227 501048007 452825681 794563043 74694797 1060844713 249966275 5592
770326101 955389515 494095244 510679878 11006784 567186686 319280631 621786623 885852581 804347807 1020333171 728411278 542868016 266280709 15065748 15006363
801302720 1116416686 847871548 48607437 1137562743 779400055 215150668 597409115 658501134 199568346 161797445 1186875624 404337506 1193833695 169468443 5963
143721783 270308531 1005158074 65500773 730951743 1059445010 432743165 662227062 1111063876 875596032 571816725 243378058 412070729 573977405 71314807 801995
460000304 722048526 266114176 1608857 438015699 559308958 286855604 914435491 70132549 612786829 819118529 1011134804 1049088202 832087187 1142707077 8571864
837194068 945674375 249407860 168752995 298358354 1014527559 611068955 637039456 404897454 985782245 12706811 261797969 507550516 250565152 625162887 5680977
371230905 413818141 393840179 304334335 390576765 574961784 536213481 643648292 482175207 1094091112 711624537 272000 1131092309 196089823 905788708 42566344
494161486 1039395429 497771244 596513866 97440260 128382914 1126423059 543790154 98585676 143841918 799784694 412669753 1107458281 663023854 188851812 515887
540317558 678975903 240558064 869397072 639139465 1101366308 414206049 52030377 620996110 304135372 816059357 599216907 364466100 819303202 771940934 1021886
562259155 1004035829 466701435 453897146 788281024 753805511 115216633 413274715 4332272 1148876902 147441489 381316131 204087819 195198972 1124891408 695650
926905877 268987151 840340176 367709032 370666257 490535811 997678254 656525810 632762260 402178827 25234235 891986026 248260243 575745952 1186699043 9034627
```

Figure 2.5: Output file for “kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt”

kruskalwithoutpq_kruskalwithpq_am00005000_input.txt

```
5000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
```

……

```
959216524 1083181559 277707042 910391103 136243075 824523169 701214058 660259220 184216167 63087992 28901359 474177105 1154213929
908621434 1098442434 315208109 1131795091 217818299 199739389 788204113 346541514 1123812718 778820237 521934658 238704747 142357439
761784075 208270207 690078598 957438041 66426565 642514880 596501668 616624418 857032055 47622120 1056597368 586273945 439913477 190834937
91804941 1079684024 16835246 698866200 359060529 776885723 1079977559 500459246 463659438 161305580 844583064 305363343 78249742
1035094232 1012161226 33211564 709977226 780761211 658988851 609697526 683362149 973057887 956871125 527302376 1022431083 423162970
47715195 64377608 1181562908 566052209 1010799237 1009028296 1045377099 80133279 648845620 897020099 436684851 844684127 522641327
368142981 324809463 641797954 127227981 728555582 696839590 468645693 824341338 560743284 250656116 529104350 21355654 303782385 781517536
135170318 206465867 139992699 386683333 622151033 388681077 693479947 126026551 621393191 71324898 887524431 445603498 987617318 325305149
806179580 640141807 479241658 708127085 1004496242 644589377 62448831 1023141431 695217071 421876417 1035081149 304233575 106598937
1022842695 360628705 525797705 882981197 67191452 521582432 838134145 732344360 504751039 192449712 40751659 1116202926 1146106705
979237369 556775160 653336703 179939452 699027820 1037961134 567938772 428321591 995679062 1183396343 48804425 10823644 269843972
585319897 155123916 953412086 543606542 63959100 1141052757 931991584 207634056 1176680415 789171591 944816708 492961878 1162149396
832457698 938982299 60828995 899013547 85393241 227312820 553379033 62672719 785998593 907934198 700608620 1104841445 948949108 22655553
575358191 754375306 376659196 1175610049 708760104 316732845 116512145 1157949007 618747869 1162898601 846883004 420796228 775037936
1063253875 13721497 829077751 178473577 64051338 711043755 672002186 1053824390 532265655 1142054509 1189491868 866300001 1113441818
879417799 1051991458 779686610 1143026193 1030096700 1019226254 455758559 452415275 815882738 59129071 119694228 740959140 259661400
744791992 36583353 701528544 712354499 976770863 658893511 162083979 273628508 1159491839 94180472 10600353 17569247 509382757 157446481
510308235 585290706 789900776 1195761739 915849278 824814365 66484776 316410812 961623098 1004311072 192899115 370310944 354563020
69820052 453857435 600607211 963156248 924292632 6754914 9995714 281282259 526759103 375632471 1100104216 856684235 558656451 384510258
755080146 1130063590 21788838 594021012 879650490 1128839767 1000875583 1046541715 1198554900 709431984 1045469698 113661938 147930055
1093496460 1064344414 603405541 822764771 790247158 487333225 476002680 1116689202 355460229 601153114 1145622282 124529848 325895050
565895690 474927340 209966442 1029820452 91102968 347270237 574879244 41741711 694232462 504402219 727240499 376492447 423129500 66272279
569967567 124517793 339928555 1054840942 1045850640 1148687046 7326429 298129975 1186526193 982993019 904495080 224163854 384564282
21846673 396570396 790362826 847746895 106648294 303439307 788301008 691587507 727743970 178244237 841885530 263545691 819484094 434124162
69448541 676793841 1191049763 943103521 910244129 634936595 187992002 1044963630 767746303 815694685 180421158 548755881 137844464
956861467 433982522 414919095 225702726 301573777 738628604 471493145 865276712 508594986 519000242 1168548808 45743886 990841790
886992747 1049523486 1107531142 972454652 34931715 351990674 251122965 719758109 410572259 605699426 1071722549 150824718 919324126
1088368750 946542864 i
```

Figure 2.6: Output file for “kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt”

kruskalwithoutpq_kruskalwithpq_am00010000_input.txt

```
10000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
```

……

```
76962766 709673646 674126576 1104579965 387378530 212804683 1023880942 916121509 572896379 716606258 365540545 241941684 727694380
261593893 681026876 852659250 127058568 403148567 759770091 664624681 947168945 458795990 863480644 948068214 364564131 326754141
745732976 632393797 379570183 6877379 362086426 19943723 226880918 48383093 535988716 775106359 702242035 1168352571 1111237901 958062541
746026213 874529386 525721513 421971276 245984585 602898810 194417772 246182787 470916290 663611414 311064342 701553964 215735448
687281777 977858060 373272247 134943839 1159591038 97743368 15195833 324460322 163600814 346767133 928714232 127532089 129090809 878140828
892578054 1070111265 282059500 515185522 572090364 219139995 382697087 118139108 672219051 465936222 400888363 1172244220 360370311
891322325 1122210363 442720789 51220097 101669946 437540906 1089037381 695214158 875619218 1100172830 1170330182 357733135 177025300
51842611 16946521 1199684930 1150271017 327212179 355739428 937179364 5872313 721364536 531676038 105650547 41997160 978129158 715600681
519074024 286406211 1049574458 726898399 296453667 911694987 702680095 747692494 748440825 32717434 1035861783 411537400 16598290
223834774 550488252 969523978 602098466 405485450 787084605 1029367906 227959921 8472820 789041727 179444435 592763283 945852700 941121635
955034428 82966890 923171904 74535383 220650708 783843355 197197635 591513717 1074631742 765921284 998202598 328091435 156839908 996177735
291233264 808105574 14502515 840838016 730382502 57806177 446895501 49742136 389800304 500683856 1166381285 184973841 616714871 136229145
1129585409 634378451 835058333 318701145 1041751439 20344543 707381282 936865889 1098546827 475326145 1011316571 1156912396 1007402429
359274918 876264879 558697435 377082145 1016246497 885659143 1138767713 122124606 379144505 843131628 536969122 670515531 340517931
755530480 550632113 1125445715 80398102 1084171069 964581155 1127605712 1105561924 1147209482 1004444316 896495934 209078658 852374383
241810718 46566231 184696796 630901354 274155572 859173829 987316963 957207615 1030405142 774224144 1174295518 708758044 900858316
507919565 558073145 524974094 530007444 108087845 657306602 563919147 692725267 951679378 1134294233 940065821 1193976365 916043320
718597062 903851217 1031463372 599397894 1136078509 526606948 29522157 1090661646 407074012 658968748 16713655 1087204557 766948797
1024089352 91001346 123476702 213054113 911036157 874813129 920000453 1153624199 359547771 460848149 1014782264 447351036 988297756
737776124 234356688 825395018 647710035 1061181119 225221941 416593532 221804602 695659290 532056682 658727077 627458203 664179993
451466716 803100111 589121249 446906430 1199588015 400049400 430247083 1195841294 213217440 979416613 1030177336 10132791 749710283
771567365 852160277 479233807 320700140 252033924 869054846 415989570 485461792 299052261 768957459 375085828 995199401 449736875
868512961 635281534 98327860 165289407 541789427 81005637 120962725 5799168 435550216 448083083 1104871621 1102740923 504048075 1060331745
203075534 87901098 954155677 376350857 279843713 1065006155 390681390 455421255 540572521 255227267 392153662 674894753 941223651
714698450 1176381967 306381294 669959619 560417299 1113343836 543839055 437630467 610102567 651674182 384095868 310520952 1035594898
27826962 1121081420 661936714 8446446 1064622035 327555824 152636641 1001633483 1040589147 798626459 393341592 835642313 776741434
1107510974 498847959 339942773 250020952 659135097 1092240437 920306313 514838081 680554374 889820064 1144324489 1087887391 869057706
1131655963 622104996 722909529 190892011 611397949 354551922 450209103 i
```

Figure 2.7: Output file for “kruskalwithoutpq_kruskalwithpq_am_00010000_input.txt”

kruskalwithoutpq_kruskalwithpq_am00100000_input.txt

```
100000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
```

……

```
99965 A99965
99966 A99966
99967 A99967
99968 A99968
99969 A99969
99970 A99970
99971 A99971
99972 A99972
99973 A99973
99974 A99974
99975 A99975
99976 A99976
99977 A99977
99978 A99978
99979 A99979
99980 A99980
99981 A99981
99982 A99982
99983 A99983
99984 A99984
99985 A99985
99986 A99986
99987 A99987
99988 A99988
99989 A99989
99990 A99990
99991 A99991
99992 A99992
99993
```

Figure 2.8: Output file for “kruskalwithoutpq_kruskalwithpq_am_00100000_input.txt”

Question 3 :

Adjacency matrix complete graphs for Kruskal algorithm without priority queue of n number of vertices for input files of different problem sizes that have been generated previously and output files with screen outputs with algorithm times for the minimum spanning tree problem. Write a function to generate all output files for each input size n.

The filenames are:

- kruskalwithoutpq_am_00000010_output.txt
- kruskalwithoutpq_am_00000100_output.txt
- kruskalwithoutpq_am_00001000_output.txt
- kruskalwithoutpq_am_00010000_output.txt
- kruskalwithoutpq_am_00100000_output.txt

Code for Output File

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    vector<Edge> edges;
};

bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

int findParent(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return findParent(parent, parent[i]);
}

void unionSet(int parent[], int x, int y) {
    int xset = findParent(parent, x);
    int yset = findParent(parent, y);
    parent[xset] = yset;
}

void kruskalMST(Graph& graph, ofstream& output) {
```

```cpp
    int V = graph.V;
    vector<Edge> result;
    int* parent = new int[V];

    // Record the start time
    auto start = chrono::system_clock::now();

    for (int i = 0; i < V; i++)
        parent[i] = i;

    sort(graph.edges.begin(), graph.edges.end(), compareEdges);

    int i = 0, e = 0;
    while (e < V - 1 && i < graph.E) {
        Edge nextEdge = graph.edges[i++];

        int x = findParent(parent, nextEdge.src);
        int y = findParent(parent, nextEdge.dest);

        if (x != y) {
            result.push_back(nextEdge);
            unionSet(parent, x, y);
            e++;
        }
    }

    // Record the end time
    auto end = chrono::system_clock::now();

    // Calculate the duration
    chrono::duration<double> duration = end - start;

    int totalWeight = 0;
```

```cpp
    output << V << endl;
    for (int i = 0; i < V; i++)
        output << i << " " << "A" << i << endl;

    for (Edge edge : result) {
        output << "A" << edge.src << " " << "A" << edge.dest << " " << edge.weight << endl;
        totalWeight += edge.weight;
    }


    output << totalWeight << endl;

    // Print the total time taken in second
    output << duration.count() << "s" << endl;

    delete[] parent;
}

int main() {
    string inputFiles[] = {
        "kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00000100_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00005000_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00010000_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00100000_input.txt"
    };

    string outputFiles[] = {
        "kruskalwithoutpq_am_00000010_output.txt",
        "kruskalwithoutpq_am_00000100_output.txt",
        "kruskalwithoutpq_am_00001000_output.txt",
        "kruskalwithoutpq_am_00005000_output.txt",
```

```
        "kruskalwithoutpq_am_00010000_output.txt",
        "kruskalwithoutpq_am_00100000_output.txt"
    };

    for (int fileIndex = 0; fileIndex < 5; fileIndex++) {
        ifstream inputFile(inputFiles[fileIndex]);

        if (!inputFile) {
            cout << "Failed to open the input file: " << inputFiles[fileIndex] << endl;
            continue;
        }

        Graph graph;
        inputFile >> graph.V;

        for (int i = 0; i < graph.V; i++) {
            int index;
            string name;
            inputFile >> index >> name;
        }

        for (int i = 0; i < graph.V; i++) {
            for (int j = 0; j < graph.V; j++) {
                int weight;
                string c;
                inputFile >> c;
                if (c == "i")
                    continue;
                else
                    weight = stoi(c);
                if (i < j) {
                    Edge edge;
                    edge.src = i;
                    edge.dest = j;
                    edge.weight = weight;
                    graph.edges.push_back(edge);
                }
            }
        }

        graph.E = graph.edges.size();

        inputFile.close();

        ofstream outputFile(outputFiles[fileIndex]);

        if (!outputFile) {
            cout << "Failed to open the output file: " << outputFiles[fileIndex] << endl;
            continue;
        }

        kruskalMST(graph, outputFile);

        outputFile.close();

        cout << "Output generated and saved to " << outputFiles[fileIndex] << "." << endl;
    }

    return 0;
}
```

Figure 3.1: Code for Question 3

```
"C:\Algorithm\Kruskal algorit
Output generated and saved to kruskalwithoutpq_am_00000010_output.txt.
Output generated and saved to kruskalwithoutpq_am_00000100_output.txt.
Output generated and saved to kruskalwithoutpq_am_00001000_output.txt.
Output generated and saved to kruskalwithoutpq_am_00005000_output.txt.
Output generated and saved to kruskalwithoutpq_am_00010000_output.txt.

Process returned 0 (0x0)   execution time : 55.287 s
Press any key to continue.
```

Figure 3.2: Screen run

Explanation

The provided code implements Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a graph. It takes input files representing graphs, constructs the graphs, and applies the algorithm to find the MST. The code utilises a disjoint-set data structure for efficient union-find operations. The MST edges, vertex names, total weight, and the execution time are then written to the output files. The code allows for processing multiple input files, making it suitable for analysing and generating MSTs for various graph instances.

Output File

kruskalwithoutpq_am_00000010_output.txt

```
10
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
A6 A7 31247930
A1 A8 34782306
A0 A7 36882129
A5 A7 63297687
A4 A5 135988061
A6 A8 149332081
A5 A9 189254917
A3 A4 220804593
A2 A3 556046821
1417636525
9.6e-06s

```

Figure 3.3: Output file for “kruskalwithoutpq_am_00000010_output.txt”

kruskalwithoutpq_am_00000100_output.txt

```
100
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A77 A99 18321524
A10 A22 18990787
A21 A91 19047753
A19 A57 19138621
A49 A54 19975349
A68 A90 20266402
A15 A42 20551122
A2 A99 20585123
A22 A24 21484476
A9 A68 21764010
A31 A91 22254194
A9 A78 22777698
A5 A66 23070590
A15 A86 24800271
A1 A87 25159352
A58 A62 25765175
A6 A67 25882446
A42 A61 26215745
A3 A4 28207002
A59 A72 30263077
A14 A75 33007184
A45 A62 33523506
A56 A92 33600667
A73 A74 40173504
A29 A54 41886420
A56 A94 44614841
A8 A83 57418144
A65 A88 66011985
A2 A53 71330156
A11 A16 77273815
1584109745
0.0019482s
```

Figure 3.4: Output file for "kruskalwithoutpq_am_00000100_output.txt"

kruskalwithoutpq_am_00001000_output.txt

```
1000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A297 A834 4284046
A136 A673 4296623
A463 A780 4307398
A470 A873 4315282
A258 A527 4340952
A238 A619 4428782
A619 A880 4613372
A21 A321 4734435
A62 A726 4883155
A145 A779 4886227
A537 A944 4979340
A101 A207 5006215
A269 A346 5009106
A143 A589 5094514
A117 A419 5679337
A106 A280 5682146
A595 A809 5720176
A576 A693 5721262
A756 A937 5759122
A61 A246 5770160
A196 A424 5823376
A378 A977 6052667
A594 A972 6211329
A18 A333 6464326
A14 A90 6561243
A496 A937 6865295
A55 A770 6913510
A0 A954 7319639
A107 A710 7478699
A154 A379 7638789
1457621244
0.145477s
```

Figure 3.5: Output file for “kruskalwithoutpq_am_00001000_output.txt”

kruskalwithoutpq_am_00005000_output.txt

```
5000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A2853 A4652 1223365
A1262 A4519 1238961
A5 A4678 1242172
A526 A3403 1249446
A648 A4349 1258524
A450 A3706 1264595
A2673 A4982 1269045
A2837 A4464 1287518
A1956 A2824 1294636
A2719 A3274 1300936
A3380 A4946 1313952
A2000 A4302 1322332
A883 A2745 1327082
A1202 A1215 1367252
A158 A1969 1368148
A1086 A3047 1413821
A2913 A4836 1433205
A1453 A2371 1471497
A1261 A4465 1484780
A905 A915 1507950
A1450 A1495 1508970
A1029 A1735 1537443
A1385 A1568 1557192
A4785 A4890 1575128
A1836 A3346 1597081
A4394 A4451 1709171
A44 A3593 1736901
A1698 A1779 1777128
A2814 A2982 1778660
A2766 A3104 2003431
1423730771
4.48969s
```

Figure 3.6: Output file for “kruskalwithoutpq_am_00005000_output.txt”

kruskalwithoutpq_am_00010000_output.txt

```
10000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A4669 A4716 727923
A1483 A2955 728038
A3636 A7027 732259
A1226 A2233 741866
A3492 A6903 746826
A3818 A5115 761652
A6185 A7165 763958
A6698 A7817 771601
A5421 A6781 774103
A895 A3179 774292
A1052 A5559 777524
A6089 A6514 787776
A6064 A6759 808667
A671 A8905 808992
A243 A7853 827225
A3241 A4893 834849
A4427 A9526 837667
A8343 A9324 841487
A60 A6956 865010
A5090 A9525 867687
A3631 A5075 896353
A6603 A8991 899077
A2053 A7029 956859
A4253 A7410 981653
A400 A7869 1083741
A5620 A8839 1091543
A1484 A3358 1116817
A4572 A9239 1125267
A5276 A7097 1267747
A2232 A9115 1385637
1457605420
19.6355s
```

Figure 3.7: Output file for “kruskalwithoutpq_am_00010000_output.txt”

Algorithm 2:Kruskal algorithm with priority queue

Question 4:

Adjacency matrix complete graphs for Kruskal algorithm with priority queue of n number of vertices for input files of different problem sizes that have been generated previously and output files with screen outputs with algorithm times for the minimum spanning tree problem. Write a function to generate all output files for each input size n.

The filenames are:
- kruskalwithpq_am_0000010_output.txt
- kruskalwithpq_am_00000100_output.txt
- kruskalwithpq_am_00001000_output.txt
- kruskalwithpq_am_00010000_output.txt
- kruskalwithpq_am_00100000_output.txt

Code for Output File

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <string>
#include <queue>
using namespace std;

// Create Edge Structure to store the data for each edge
struct Edge {
    int src, dest, weight;
};


// Create Graph Structure to store the data for each verticies and edges
struct Graph {
    int V, E;
    vector<Edge> edges;
};

// To sort
bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

struct CompareEdges{
    bool operator()(Edge const& edge1, Edge const& edge2) {
        return edge1.weight > edge2.weight;
    };
};

// Find Parent Node? I assuming Root Node
```

```cpp
int findParent(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return findParent(parent, parent[i]);
}

// Assuming is for adjacency matrix
void unionSet(int parent[], int x, int y) {
    int xset = findParent(parent, x);
    int yset = findParent(parent, y);
    parent[xset] = yset;
}

// Calculating Kruskal
void kruskalMST(Graph& graph, ofstream& output) {
    int V = graph.V;
    vector<Edge> result;
    int* parent = new int[V];

    // Record the start time
    auto start = chrono::system_clock::now ();

    for (int i = 0; i < V; i++)
        parent[i] = i;

    // Using Priority Queue to sort
    priority_queue<Edge, vector<Edge>, CompareEdges> pq;

    for (Edge e: graph.edges){
        pq.emplace(e);
    }
    graph.edges.clear();
    while (!pq.empty()) {
```

```cpp
        Edge e = pq.top();
        pq.pop();
        graph.edges.push_back(e);
    }

    // sort(graph.edges.begin(), graph.edges.end(), compareEdges);

    int i = 0, e = 0;
    while (e < V - 1 && i < graph.E) {
        Edge nextEdge = graph.edges[i++];

        int x = findParent(parent, nextEdge.src);
        int y = findParent(parent, nextEdge.dest);

        if (x != y) {
            result.push_back(nextEdge);
            unionSet(parent, x, y);
            e++;
        }
    }

    // Record the end time
    auto end = chrono::system_clock::now ();

    // Calculate the duration
    chrono::duration < double >duration = end - start;

    int totalWeight = 0;
    output << V << endl;
    for (int i = 0; i < V; i++)
        output << i << " " << "A" << i << endl;

    for (Edge edge : result) {
```

```
        output << "A" << edge.src << " " << "A" << edge.dest << " " << edge.weight << endl;
        totalWeight += edge.weight;
    }

    output << totalWeight << endl;

    // Print the total time taken in second
    output << duration.count () << "s" << endl;

    delete[] parent;
}

int main() {
    string inputFiles[] = {
        "kruskalwithoutpq_kruskalwithpq_am_00000010_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00000100_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00001000_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00005000_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00010000_input.txt",
        "kruskalwithoutpq_kruskalwithpq_am_00100000_input.txt"
    };

    string outputFiles[] = {
        "kruskalwithpq_am_00000010_output.txt",
        "kruskalwithpq_am_00000100_output.txt",
        "kruskalwithpq_am_00001000_output.txt",
        "kruskalwithpq_am_00005000_output.txt",
        "kruskalwithpq_am_00010000_output.txt",
        "kruskalwithpq_am_00100000_output.txt"
    };

    for (int fileIndex = 0; fileIndex < 5; fileIndex++) {
        ifstream inputFile(inputFiles[fileIndex]);

        if (!inputFile) {
            cout << "Failed to open the input file: " << inputFiles[fileIndex] << endl;
            continue;
        }

        Graph graph;
        inputFile >> graph.V;

        for (int i = 0; i < graph.V; i++) {
            int index;
            string name;
            inputFile >> index >> name;
        }

        for (int i = 0; i < graph.V; i++) {
            for (int j = 0; j < graph.V; j++) {
                int weight;
                string c;
                inputFile >> c;
                if (c == "i")
                    continue;
                else
                    weight = stoi(c);
                if (i < j) {
                    Edge edge;
                    edge.src = i;
                    edge.dest = j;
                    edge.weight = weight;
                    graph.edges.push_back(edge);
                }
            }
        }
```

```cpp

        graph.E = graph.edges.size();

        inputFile.close();

        ofstream outputFile(outputFiles[fileIndex]);

        if (!outputFile) {
            cout << "Failed to open the output file: " << outputFiles[fileIndex] << endl;
            continue;
        }

        kruskalMST(graph, outputFile);

        outputFile.close();

        cout << "Output generated and saved to " << outputFiles[fileIndex] << "." << endl;
    }

    return 0;
}
```

Figure 4.1: Code for Question 4

```
"C:\Algorithm\Kruskal algorit
Output generated and saved to kruskalwithpq_am_00000010_output.txt.
Output generated and saved to kruskalwithpq_am_00000100_output.txt.
Output generated and saved to kruskalwithpq_am_00001000_output.txt.
Output generated and saved to kruskalwithpq_am_00005000_output.txt.
Output generated and saved to kruskalwithpq_am_00010000_output.txt.

Process returned 0 (0x0)   execution time : 137.363 s
Press any key to continue.
```

Figure 4.2: Screen run

Explanation

The provided code implements Kruskal's algorithm to find the minimum spanning tree (MST) of a graph. It reads input graphs from files, sorts the edges using a priority queue, and then iterates through the edges to construct the MST by merging disjoint sets. The MST details, including the total weight and execution time, are written to output files. This code efficiently computes the MST for multiple test cases and utilizes priority queue for edge sorting, making it a practical implementation of Kruskal's algorithm.

Output File

kruskalwithpq_am_00000010_output.txt

```
10
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
A6 A7 31247930
A1 A8 34782306
A0 A7 36882129
A5 A7 63297687
A4 A5 135988061
A6 A8 149332081
A5 A9 189254917
A3 A4 220804593
A2 A3 556046821
1417636525
2.03e-05s
```

Figure 4.3: Output file for "kruskalwithpq_am_0000010_output.txt"

kruskalwithpq_am_00000100_output.txt

```
100
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A77 A99 18321524
A10 A22 18990787
A21 A91 19047753
A19 A57 19138621
A49 A54 19975349
A68 A90 20266402
A15 A42 20551122
A2 A99 20585123
A22 A24 21484476
A9 A68 21764010
A31 A91 22254194
A9 A78 22777698
A5 A66 23070590
A15 A86 24800271
A1 A87 25159352
A58 A62 25765175
A6 A67 25882446
A42 A61 26215745
A3 A4 28207002
A59 A72 30263077
A14 A75 33007184
A45 A62 33523506
A56 A92 33600667
A73 A74 40173504
A29 A54 41886420
A56 A94 44614841
A8 A83 57418144
A65 A88 66011985
A2 A53 71330156
A11 A16 77273815
1584109745
0.0022266s

```

Figure 4.4: Output file for “kruskalwithpq_am_0000100_output.txt”

kruskalwithpq_am_00001000_output.txt

```
1000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A297 A834 4284046
A136 A673 4296623
A463 A780 4307398
A470 A873 4315282
A258 A527 4340952
A238 A619 4428782
A619 A880 4613372
A21 A321 4734435
A62 A726 4883155
A145 A779 4886227
A537 A944 4979340
A101 A207 5006215
A269 A346 5009106
A143 A589 5094514
A117 A419 5679337
A106 A280 5682146
A595 A809 5720176
A576 A693 5721262
A756 A937 5759122
A61 A246 5770160
A196 A424 5823376
A378 A977 6052667
A594 A972 6211329
A18 A333 6464326
A14 A90 6561243
A496 A937 6865295
A55 A770 6913510
A0 A954 7319639
A107 A710 7478699
A154 A379 7638789
1457621244
0.337052s
```

Figure 4.5: Output file for "kruskalwithpq_am_0001000_output.txt"

kruskalwithpq_am_00005000_output.txt

```
5000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A2853 A4652 1223365
A1262 A4519 1238961
A5 A4678 1242172
A526 A3403 1249446
A648 A4349 1258524
A450 A3706 1264595
A2673 A4982 1269045
A2837 A4464 1287518
A1956 A2824 1294636
A2719 A3274 1300936
A3380 A4946 1313952
A2000 A4302 1322332
A883 A2745 1327082
A1202 A1215 1367252
A158 A1969 1368148
A1086 A3047 1413821
A2913 A4836 1433205
A1453 A2371 1471497
A1261 A4465 1484780
A905 A915 1507950
A1450 A1495 1508970
A1029 A1735 1537443
A1385 A1568 1557192
A4785 A4890 1575128
A1836 A3346 1597081
A4394 A4451 1709171
A44 A3593 1736901
A1698 A1779 1777128
A2814 A2982 1778660
A2766 A3104 2003431
1423730771
15.8326s

```

Figure 4.6: Output file for “kruskalwithpq_am_0005000_output.txt”

kruskalwithpq_am_00010000_output.txt

```
10000
0 A0
1 A1
2 A2
3 A3
4 A4
5 A5
6 A6
7 A7
8 A8
9 A9
10 A10
11 A11
12 A12
13 A13
14 A14
15 A15
16 A16
17 A17
18 A18
19 A19
20 A20
21 A21
22 A22
23 A23
24 A24
25 A25
26 A26
27 A27
28 A28
29 A29
30 A30
31 A31
```

……

```
A4669 A4716 727923
A1483 A2955 728038
A3636 A7027 732259
A1226 A2233 741866
A3492 A6903 746826
A3818 A5115 761652
A6185 A7165 763958
A6698 A7817 771601
A5421 A6781 774103
A895 A3179 774292
A1052 A5559 777524
A6089 A6514 787776
A6064 A6759 808667
A671 A8905 808992
A243 A7853 827225
A3241 A4893 834849
A4427 A9526 837667
A8343 A9324 841487
A60 A6956 865010
A5090 A9525 867687
A3631 A5075 896353
A6603 A8991 899077
A2053 A7029 956859
A4253 A7410 981653
A400 A7869 1083741
A5620 A8839 1091543
A1484 A3358 1116817
A4572 A9239 1125267
A5276 A7097 1267747
A2232 A9115 1385637
1457605420
91.2588s
```

Figure 4.7: Output file for “kruskalwithpq_am_0010000_output.txt”

Algorithm 3:Huffman Algorithm

Question 5:

Huffman coding of n number of words implementation for file inputs and file outputs with screen outputs with step-by-step illustration for the lossless data compression problem.

Input filename : huffmancoding_00000003_input.txt	description	Output filename: huffmancoding_00000003_input.txt	description

6
A
B
C
G
K
T
CBKTG CACGA
GCTA
	// num of unique char
	// character, one character is 7-bit
	
	// num of word with space
		6
		A 3 00
		B 1 1110
		C 4 10
		G 3 01
		K 1 1111
		T 2 110
		34 bits out of 98 bits
		Total of 34%
		10s
			// num of unique char
			// character, frequencies, code words, character bits, in alphabetical order.
			
			//num of total bits
			//total space int percentage
			// example total time taken

Code for Output File

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <map>
#include <chrono>
#include <queue>
#include <unordered_map>

using namespace std;

struct Node {
    char character;
    int frequency;
    Node* left;
    Node* right;

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

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->frequency > b->frequency;
    }
};

void encode(Node* root, string code, unordered_map<char, string>& encoding) {
    if (root == nullptr)
        return;
```

Figure 5.1 : OutputFile code part 1

```
    // Leaf node
    if (!root->left && !root->right) {
        encoding[root->character] = code;
        return;
    }

    // Recursive calls for left and right subtrees
    encode(root->left, code + "0", encoding);
    encode(root->right, code + "1", encoding);


nordered_map<char, string> buildHuffmanTree(string text) {
    unordered_map<char, int> frequency;
    for (char ch : text)
        frequency[ch]++;

    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Create a leaf node for each character and add it to the priority queue
    for (auto& pair : frequency) {
        Node* newNode = new Node(pair.first, pair.second);
        pq.push(newNode);
    }

    // Build the Huffman tree
    while (pq.size() > 1) {
        Node* left = pq.top();
        pq.pop();

        Node* right = pq.top();
```

Figure 5.2 : OutputFile code part 2

```
        pq.pop();

        Node* right = pq.top();
        pq.pop();

        Node* combined = new Node('$', left->frequency + right->frequency);
        combined->left = left;
        combined->right = right;

        pq.push(combined);
    }

    Node* root = pq.top();
    unordered_map<char, string> encoding;
    encode(root, "", encoding);

    delete root;
    return encoding;
}

map<char, int> calculateCharacterFrequencies(const string& str) {
    map<char, int> frequencies;
    for (char c : str) {
        if (isalpha(c)) {  // Count only alphabetical characters
            frequencies[c]++;
        }
    }
    return frequencies;
}
```

Figure 5.3 : OutputFile code part 3

```
int main() {
    // Declare the startTime variable
    std::chrono::steady_clock::time_point startTime = std::chrono::steady_clock::now();

    string inputFileName = "group205_num05_huffmancoding_00000003_input.txt";
    string outputFileName = "group205_num05_huffmancoding_00000003_output.txt";

    ifstream inputFile(inputFileName);
    ofstream outputFile(outputFileName);

    if (inputFile.is_open()) {
        string line;
        string lastLine;
        while (getline(inputFile, line)) {
            lastLine = line;
        }
        inputFile.close();

        // Remove spaces from the last line
        lastLine.erase(remove_if(lastLine.begin(), lastLine.end(), [](char c) { return isspace(c); }), lastLine.end());

        // Calculate character frequencies and build Huffman tree
        map<char, int> frequencies = calculateCharacterFrequencies(lastLine);
        unordered_map<char, string> encoding = buildHuffmanTree(lastLine);

        // Print character frequencies and encodings
        int totalBits = 0;
        int bit = 7;
        int totalFrequency = 0;
```

Figure 5.4 : OutputFile code part 4

```cpp
    for (const auto& pair : frequencies) {
        totalFrequency += pair.second;
    }

    // Write the number of unique characters to the output file
    int numUniqueChars = frequencies.size();
    outputFile<< numUniqueChars << endl;

    for (const auto& pair : frequencies) {
        char character = pair.first;
        int frequency = pair.second;
        string code = encoding[character];
        int codeLength = code.length();
        int frequencyWithCode = frequency * codeLength;
        outputFile << character << " " << frequency << " " << code << " " << frequencyWithCode << endl;
        totalBits += frequencyWithCode;
    }


    // Calculate the percentage of words with spaces
    double percentageWithSpaces = (static_cast<double>(totalBits));

    // Calculate and print the program execution time
    std::chrono::steady_clock::time_point endTime = std::chrono::steady_clock::now();
    std::chrono::duration<double> duration = std::chrono::duration_cast<std::chrono::duration<double>>(endTime - startTime);
    std::string executionTime = std::to_string(duration.count()) + "s";
    outputFile << totalBits << "-bits out of-" << totalFrequency * bit << "bit" << endl;
    outputFile <<"Total space "<< percentageWithSpaces << "%" << endl;
    outputFile << executionTime << endl;
```

Figure 5.5 : OutputFile code part 5

```cpp
        outputFile.close();

        cout << "Output has been written to '" << outputFileName << "'." << endl;
    } else {
        cout << "Failed to open the file." << endl;
    }

    return 0;
}
```

Figure 5.6 : OutputFile code part 6

```
C:\Users\tuf\Downloads\TCP_Assignment\group205_num05_huffmancoding_00000003_output.exe
Output has been written to 'group205_num05_huffmancoding_00000003_output.txt'.

Process returned 0 (0x0)   execution time : 0.110 s
Press any key to continue.
```

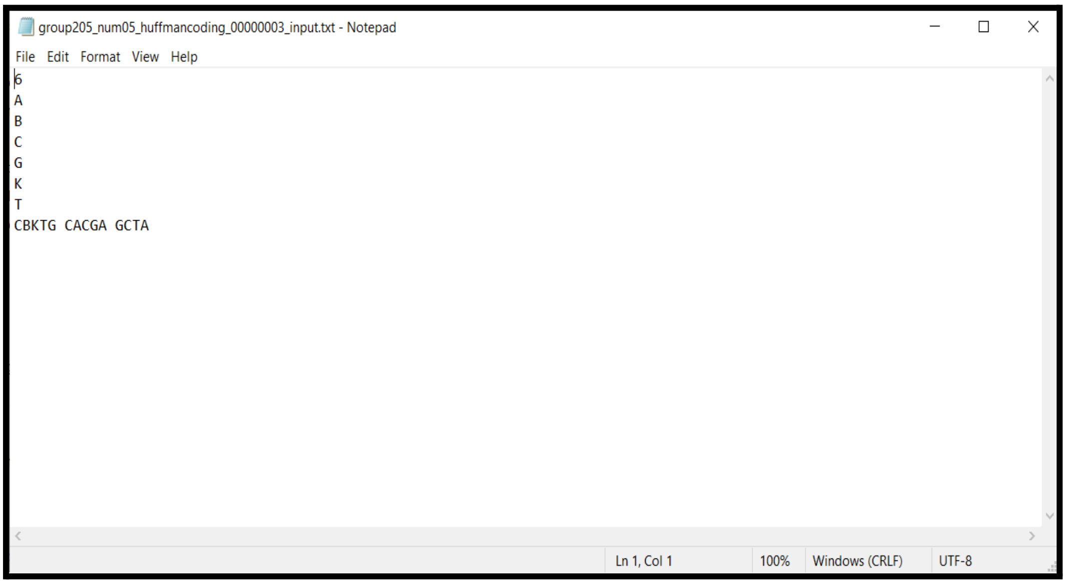
Figure 5.7: Command prompt for input file

Explanation

The provided code implements the Huffman coding algorithm, a lossless data compression technique. It begins by including the necessary header files and defining structures for the Huffman tree. The encode function assigns binary codes to each character in the tree, while the buildHuffmanTree function constructs the tree based on character frequencies. The calculateCharacterFrequencies function determines the frequency of each character. In the main function, the code reads a text file, calculates character frequencies, builds the Huffman tree, and generates the character encodings. It then writes statistics such as the number of unique characters, character frequencies, encoding details, total bits used, percentage of space saved, and execution time to an output file. Overall, the code performs Huffman coding to compress the text and provides insights into the compression achieved.

Output File

group205_num05_huffmancoding_00000003_input.txt




```
6
A
B
C
G
K
T
CBKTG CACGA GCTA
```

Figure 5.8: Input file for sample .

Question 6:

Random words for Huffman coding algorithm of n number of words implementation for dataset generation of input files that contain random words in each file (10 words, 100 words, 1000 words, 10000 words, 100000 words, etc.). Write a function to generate all the input files. The filenames are:

- huffmancoding_00000010_input.txt
- huffmancoding_00000100_input.txt
- huffmancoding_00001000_input.txt
- huffmancoding_00010000_input.txt
- huffmancoding_00100000_input.txt

Code for Input File

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <iomanip>
#include <sstream>

using namespace std;

// Function to generate random words
string generateRandomWord(int wordLength) {
    string word;
    static const char alphabet[] =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for (int i = 0; i < wordLength; ++i) {
        word += alphabet[rand() % (sizeof(alphabet) - 1)];
    }

    return word;
}
```

Figure 6.1 : InputFile code part 1

```cpp
// Function to generate an input file with random words
void generateInputFile(const string& filename, int numCharacters, int charBitLength) {
    ofstream outputFile(filename);

    if (!outputFile) {
        cerr << "Failed to create file: " << filename << endl;
        return;
    }

    // Calculate the number of words required to achieve the desired number of characters
    int wordLength = charBitLength;
    int numWords = numCharacters / wordLength;
    int remainingCharacters = numCharacters % wordLength;

    // Generate the random words
    vector<string> words;
    for (int i = 0; i < numWords; ++i) {
        string word = generateRandomWord(wordLength);
        words.push_back(word);
    }

    if (remainingCharacters > 0) {
        string word = generateRandomWord(remainingCharacters);
        words.push_back(word);
    }

    // Find unique characters
    string allWords = "";
    for (const string& word : words) {
        allWords += word;
    }
    sort(allWords.begin(), allWords.end());
    auto last = unique(allWords.begin(), allWords.end());
    allWords.erase(last, allWords.end());
```

Figure 6.2 : InputFile code part 2

```cpp
    // Write the number of unique characters
    int numUniqueChars = allWords.length();
    outputFile << numUniqueChars << endl;

    // Write the unique characters and their bit lengths
    for (char ch : allWords) {
        outputFile << ch << endl;
    }

    // Write the generated words with spaces every 5 characters
    int count = 0;
    for (const string& word : words) {
        for (char ch : word) {
            outputFile << ch;
            ++count;
            if (count % 5 == 0) {
                outputFile << " ";
            }
        }
    }

    outputFile.close();
    cout << "Generated file: " << filename << endl;
}


int main() {
    // Set the seed for random number generation
    srand(1201101003);

    // Number of characters for each input file
    vector<int> characterCounts = {10, 100, 1000, 10000, 100000};

    int charBitLength = 7;

    for (int count : characterCounts) {
        ostringstream oss;
        oss << "group205_num06_huffmancoding_" << setfill('0') << setw(8) << count << "_input.txt";
        string filename = oss.str();
        generateInputFile(filename, count, charBitLength);
    }
```

Figure 6.3 : InputFile code part 3

```cpp
    }

    return 0;
}
```

Figure 6.4 : InputFile code part 4

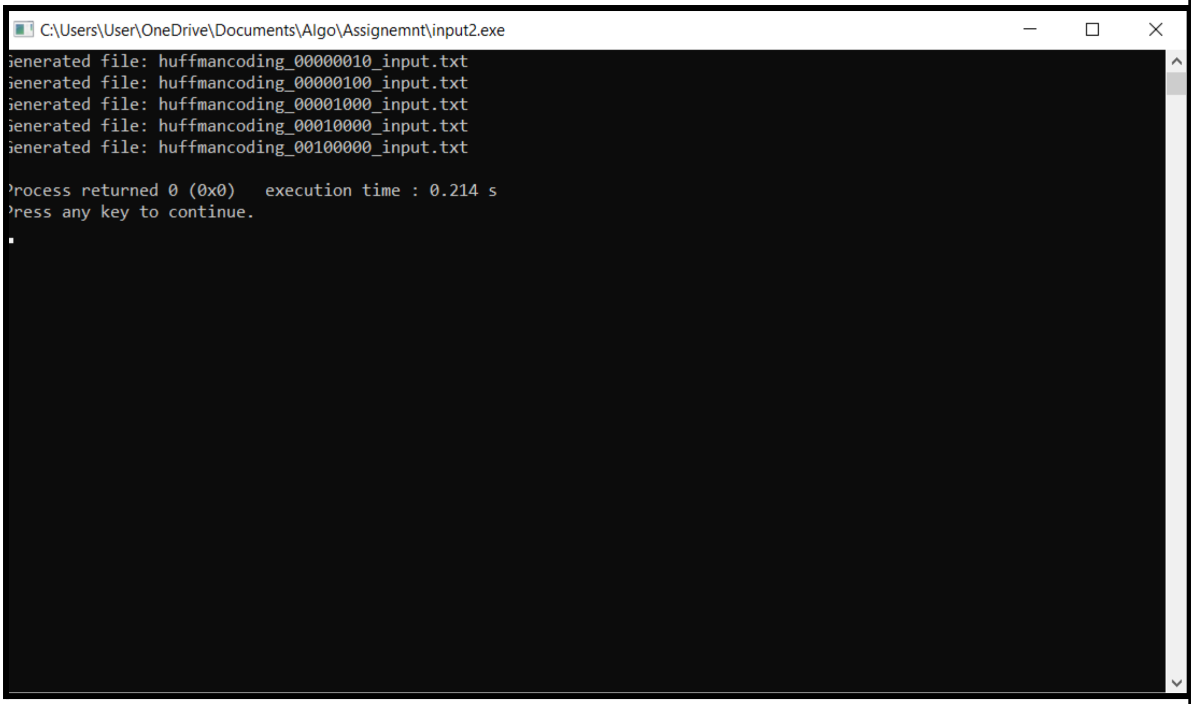


Figure 6.5: Command prompt for input file

Explanation

The provided code is a program that generates input files for Huffman coding. It consists of several functions and a main function. The "generateRandomWord" function generates a random word of a specified length using uppercase alphabets. The "generateInputFile" function creates an output file for Huffman coding, taking parameters such as the filename, the desired total number of characters, and the number of bits used to represent each character. Inside the "generateInputFile" function, an output file stream is created, and if it fails to open, an error message is printed. The function calculates the number of words required to achieve the desired character count and initialises a vector to store the generated random words. A loop is used to generate random words of the specified length, and if there are remaining characters, an additional word is generated. The unique characters from all the generated words are extracted, and the number of unique characters is written to the output file. The unique characters and the generated words are then written to the output file with specific formatting. Finally, the output file is closed, and a message is printed indicating the filename of the generated file. In the main function, the random number generator is seeded, and a vector is defined to store the desired character counts for the input files. The generateInputFile function is called in a loop for each count, generating different files. After the loop, the main function returns 0 to indicate successful execution.

Input File

huffmancoding_00000010_input.txt:

huffmancoding_00000010_input.txt - Notepad

File Edit Format View Help

```
9
C
D
I
N
Q
S
V
Y
Z
NIYQZ VSZCD
```

Figure 6.6: Input file for 10 characters.

huffmancoding_00000100_input.txt:

huffmancoding_00000100_input.txt - Notepad

File Edit Format View Help

```
26
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
JALAU PVLKL JZYUG IHKNX WOWHF SYOVT QCQUL SNCLM MVLSC RIVQK VBJQI CAWJV XCNKS KZEDP FHFYE ZCAKD DWNSJ IVBFK JJFQI UFYCA
```

Figure 6.7: Input file for 100 characters.

huffmancoding_00001000_input.txt:

```
*huffmancoding_00001000_input.txt - Notepad
File  Edit  Format  View  Help
26
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
KFEDO BIFIY JIZVO ONSSN DSMZS YLZSB UZXSA MWAJU RVJFP JYWDF UWZKA FGWOE MJPWB WOQTW ONDBJ QFVDS GIXAR MANPA GMBVL
VDBMY ZXZNZ GNKZA ANNZJ IDMAI RCTEM KQADH AMNMI TOUXE NKPNY QGVLJ VWCAU GQXWM DGZXE XWUMF UXPLO VYERL CFPYG DNRNF
FDWOP MWBPA EVOIN YDUOW YUMIQ HCEMD CMUSR DUQUO FKEVH EUUDI UARZV IPKBF AOKNK FGGIS POVCJ JEGTF RELVZ MGVQU KFNPR
RUQCJ YHBPV YOQOU NKDIO VKKQB OMFEN HACQZ OJPCH TSLZT MZUBD TNUNA TOVJW IXERT LYLTZ XKFUW FKJED AFVMO MFFNJ FUPDW
VANCT FETCO LWDTW IEHMF ONRAN RTBIF PMRDW ESXAE QASMZ TIGOQ BMETJ KPOFC HIECT CFNBE CDRMP MABCZ OZGJD GOOFV DLKKW
NACWJ ZSNOD AHONJ LKESZ FFLHO TQPLB QSFGJ JBFCB LKUAJ AOHKB KUWYE EHANM PKIQG WAOIH SQULM UTGWA YNPSB MCTIJ YBSGQ
MLHHU PSLGM ZOARJ NUIRE XVPMT QOBMU AEPXY FKBYV OPRYY MMRUH KSXPX UQGBP DIOQV APGSQ GSOSN LUXUM COEXH VOQIF KQAKR
YIOSC OVTQK SPNZJ JASII JWFST KIYDS REKTK MMAIJ SVPRN ZUETN UHJUV HOADN NKJPF DCQXV WHIYV KIAXF RCVBA HLFFI DXIOW
COAPJ NZFTX JSOUJ CXNXM DISAQ BXQBG OKORY ODLQC ADAKV BORNF QTEIX SEUDZ ABSYF HHAFR REZTR MJAQZ WDOJI YQGCK LOSLV
MNOVU IHVXO ZMLHQ JDOTR VZUZW RVDCX LNQME RACFU IKDJI TLZGT JJGSA NMLZB UYGCQ XFOXZ VMBKM SDBKR XMDDP EQFRV USWGQ
HLKSU RQRUX WDPFV XZIEP ZDYIF WQUGN ESABY XXJEJ MYDSQ BUWOT
```

Figure 6.8: Input file for 1000 characters.

huffmancoding_00010000_input.txt:

```
huffmancoding_00010000_input.txt - Notepad
File  Edit  Format  View  Help
26
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
GWVFS SPBYU WBVUU AKDKZ YJQBU QJMNZ MZMAE WVLTI AGIXN QLQDY BYKHO TNBPT BKPAJ DGFGZ WQWPI DCURW HPWBY ODRYJ LUSLM JYWJU RRYLN YVMHR LMYOK NKYWM EQFWW OFDTZ WCVYZ KYPBF PBJP
U INYZK CFQTA SRXNS DAHKB VGJKN OHBDO VTKOR UBGYV XIRQL FATBZ FYHXV MNFRB PGDZH IWYOU QVMJG WQOQS OGNPJ CLXIS QGOWN WFKNL XSPXT FRNDO MUQRG QOOFS IKWQG YWWZK KSVYY VQELE VT
BYP EQHDW NPELQ KZTDP YFTNN MXHJB MQFCW HOQPY VUPVD FRQWW TDQRK CONIK GJPXS NECRA PHGIO HRANU EMEHJ CTCNR MEQAW KFDQL DCRNW ZCOLG ADPEN HJTUB IZPTK ZFJLE UMYGM VWJKS GBVDG
REVWY GJTZD VWNPY KOAHT AVMUK GWWNC AJBLD HDHXL SMNDQ KBILI RBDLE CYHSI EEGNK MYHZE CSMHE AWHWV RBDVK CXKHW SNWSK RSFUX ZUOGT CVEKU NEZGA SVNPA VWNAN YMUYL EUAPQ ELVFY XOUR
S EXCQP XEARO PLIUW BLHGW ZEFZT NSWWZ IQSPJ AJASG JIVVM MTLJF GVONS BBXJS NELFQ IUNRC SVQQJ TDIUE VLDQN BFDXL JKLNH OCXJR JRQWK CPPEV TZCLH YNKHT OPXFC CUCTK GRQOR ZMZET GK
WSU AZGBK EGNMV PJJZR AUUVN WGYCD SJWOO TTANS VGHBJ LSYJB SXLDO PNYDE RXJFB ZGXBN OMTDT SAHVB PGKHP DWLHZ FJKAS PSGHC EEHHE CCOJN HBXXS CUXYO XUROQ MGONU LLMFF BKKJV KFFOE
TCPMV VQXFQ FNCUL FESUD KLYUY XYTTR QBVIG ZBCCA VSZFY CMMWB IUUGN EPUQC LQBAP MYQZW JQTMR VUOQY JCVXS EUUJE VEUWQ RGAHH EESXI YOCPW QTGHN PKEYP WTVKP LDVFP QDSLP EFOGP XMIG
Y XPVZE HWETK UKWSF BOKQY EJNSS OKMDS CYVWW SMHAF KVVZC KNYJU RRPZN PGHWX HAOZE WHAHW VVIAE SLINO MPZVD UUUNM BOOSQ PGECX HLWXH TBFEZ LXIKV UIQSH FTJOE AEJMU UKQEF HGDGQ OU
ALI TFJKT NXARL TCVWY WHXSX RAQUA ENTLH BUZIO ERJPF OCMBO ZNPHL PSNHX DTSYL GIWNB KRDJA RFGPE RSRAP QCGCN TDJBE LOGQO SBMYM YWAZP ZBTUX GGHOC CQGMB MXZTT NCCBN OLSCK BYTZF
CGMZL FDNFX KSIRH EWGOD UPRLK ZBEFL IBVYY QQMGB IHYND SQVZN KVJTY WCFQK OLAVJ UOIMA CTJVD DMPHY JICHG MOFPG EUEVU RHRMS MXDCD MJYDO AVDDJ EDXEF AYSLI XEGXB GNVUX AVVCN XXHB
D MPNAN HXCJX HPHRR URZWX MUXOI JZOOL TJNJV OICPS ZORLN XYXTC MGBOH AOJKP PUSHZ OVTIC ILGGR MIXBX NKCVE YWDOD ARFTP YHXZC KGASJ FBRRO MEYIS XPGNY GLHQE KSTVG ZPLNE DPZLO DH
UUE RRYOP YGHID EVDIL DTCNZ DVPHL OKGJI GQHUT CSNBB QZOPR HZQLL LDHGN SGZQZ ZJZZA OWIPJ WOFCT EOPXC XDOAZ ASZEY PZXIU VVAGN ZKIHV EVUVF UHXAA FDGXN QOPEL ZCUON PBYZT GFKVA

Ln 1, Col 1    110%    Windows (CRLF)    UTF-8
```

Figure 6.9: Input file for 10000 characters.

huffmancoding_00100000_input.txt:

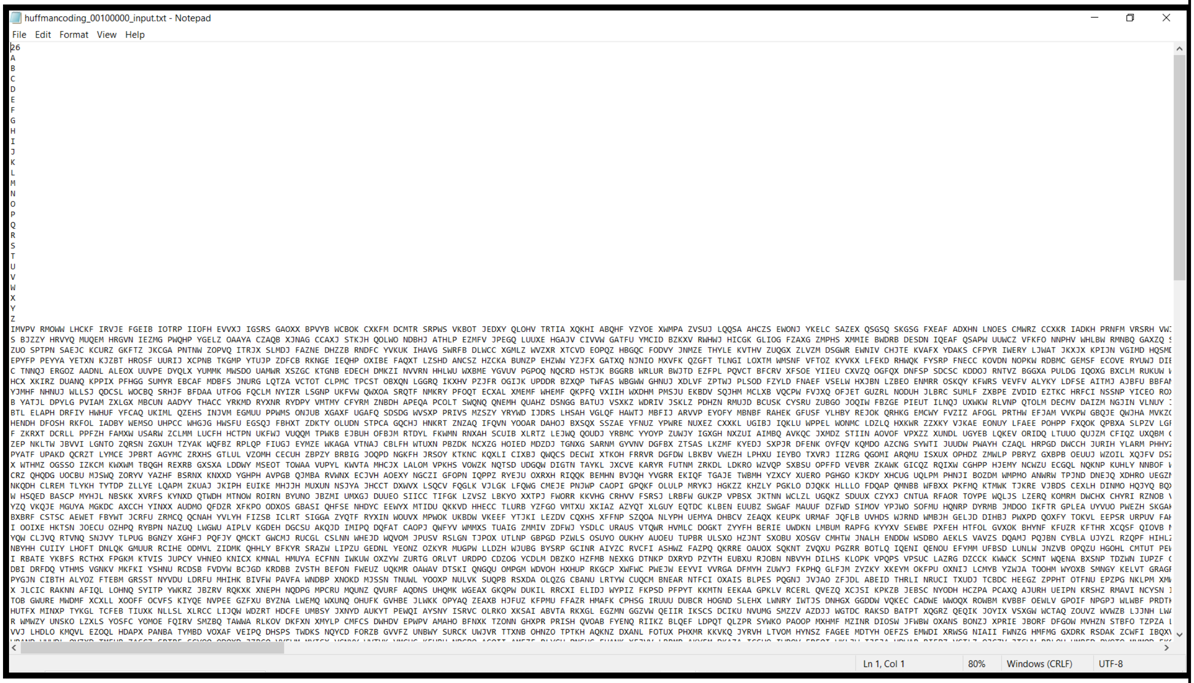


Figure 6.10: Input file for 100000 characters.

Question 7:

Huffman coding algorithm of n number of words for input files of different problem sizes that have been generated previously and output files with screen outputs with algorithm space percentages for the lossless data compression problem. Write a function to generate all output files for each input size n.

- huffmancoding_00000010_output.txt
- huffmancoding_00000100_output.txt
- huffmancoding_00001000_output.txt
- huffmancoding_00010000_output.txt
- huffmancoding_00100000_output.txt

Code for Output File

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <map>
#include <chrono>
#include <queue>
#include <unordered_map>

using namespace std;

struct Node {
    char character;
    int frequency;
    Node* left;
    Node* right;

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

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->frequency > b->frequency;
    }
};

void encode(Node* root, string code, unordered_map<char, string>& encoding) {
    if (root == nullptr)
        return;
```

Figure 6.1 : InputFile code part 1

```
    // Leaf node
    if (!root->left && !root->right) {
        encoding[root->character] = code;
        return;
    }

    // Recursive calls for left and right subtrees
    encode(root->left, code + "0", encoding);
    encode(root->right, code + "1", encoding);
}

unordered_map<char, string> buildHuffmanTree(string text) {
    unordered_map<char, int> frequency;
    for (char ch : text)
        frequency[ch]++;

    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Create a leaf node for each character and add it to the priority queue
    for (auto& pair : frequency) {
        Node* newNode = new Node(pair.first, pair.second);
        pq.push(newNode);
    }

    // Build the Huffman tree
    while (pq.size() > 1) {
        Node* left = pq.top();
        pq.pop();
```

Figure 6.2 : InputFile code part 2

```
        Node* combined = new Node('$', left->frequency + right->frequency);
        combined->left = left;
        combined->right = right;

        pq.push(combined);
    }

    Node* root = pq.top();
    unordered_map<char, string> encoding;
    encode(root, "", encoding);

    delete root;
    return encoding;
}

map<char, int> calculateCharacterFrequencies(const string& str) {
    map<char, int> frequencies;
    for (char c : str) {
        if (isalpha(c)) {  // Count only alphabetical characters
            frequencies[c]++;
        }
    }
    return frequencies;
}

int main() {
    // Declare the startTime variable
    std::chrono::steady_clock::time_point startTime = std::chrono::steady_clock::now();
```

Figure 6.3 : InputFile code part 3

```
string inputFileName = "huffmancoding_00100000_input.txt";
string outputFileName = "huffmancoding_00100000_output.txt";

ifstream inputFile(inputFileName);
ofstream outputFile(outputFileName);

if (inputFile.is_open()) {
    string line;
    string lastLine;
    while (getline(inputFile, line)) {
        lastLine = line;
    }
    inputFile.close();

    // Remove spaces from the last line
    lastLine.erase(remove_if(lastLine.begin(), lastLine.end(), [](char c) { return isspace(c); }), lastLine.end());

    // Calculate character frequencies and build Huffman tree
    map<char, int> frequencies = calculateCharacterFrequencies(lastLine);
    unordered_map<char, string> encoding = buildHuffmanTree(lastLine);

    // Print character frequencies and encodings
    int totalBits = 0;
    int bit = 7;
    int totalFrequency = 0;

    for (const auto& pair : frequencies) {
        totalFrequency += pair.second;
    }
```

Figure 6.4 : InputFile code part 4

```
    // Write the number of unique characters to the output file
    int numUniqueChars = frequencies.size();
    outputFile<< numUniqueChars << endl;

    for (const auto& pair : frequencies) {
        char character = pair.first;
        int frequency = pair.second;
        string code = encoding[character];
        int codeLength = code.length();
        int frequencyWithCode = frequency * codeLength;
        outputFile << character << " " << frequency << " " << code << " " << frequencyWithCode << endl;
        totalBits += frequencyWithCode;
    }


    // Calculate the percentage of words with spaces
    double percentageWithSpaces = (static_cast<double>(totalBits));

    // Calculate and print the program execution time
    std::chrono::steady_clock::time_point endTime = std::chrono::steady_clock::now();
    std::chrono::duration<double> duration = std::chrono::duration_cast<std::chrono::duration<double>>(endTime - startTime);
    std::string executionTime = std::to_string(duration.count()) + "s";
    outputFile << totalBits << "-bits out of-" << totalFrequency * bit << "bit" << endl;
    outputFile <<"Total space "<< percentageWithSpaces << "%" << endl;
    outputFile << executionTime << endl;


    outputFile.close();
```

Figure 6.5 : InputFile code part 5

```
        cout << "Output has been written to '" << outputFileName << "'." << endl;
    } else {
        cout << "Failed to open the file." << endl;
    }

    return 0;
}
```

Figure 6.6 : InputFile code part 5

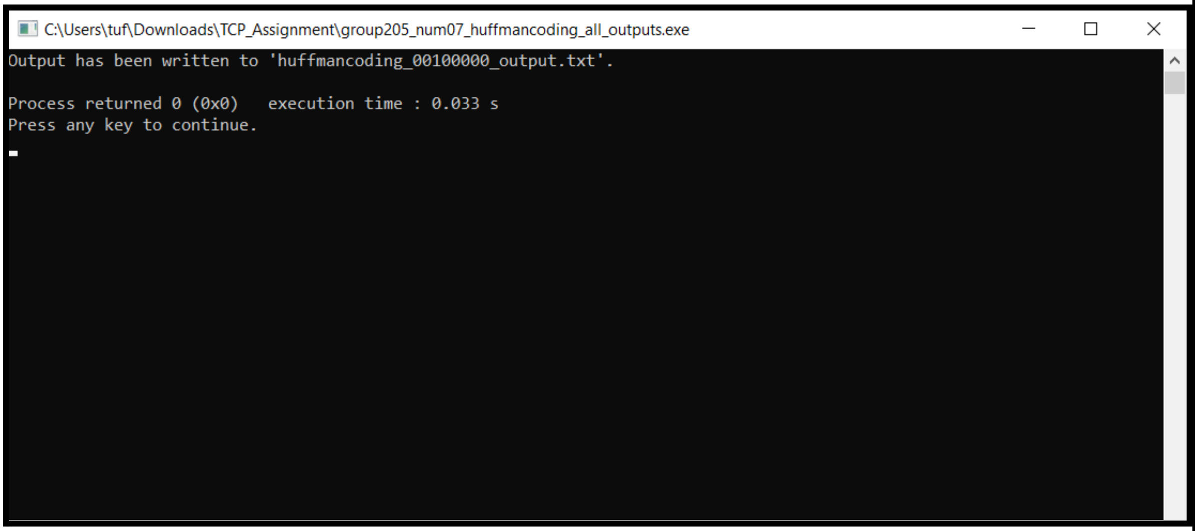


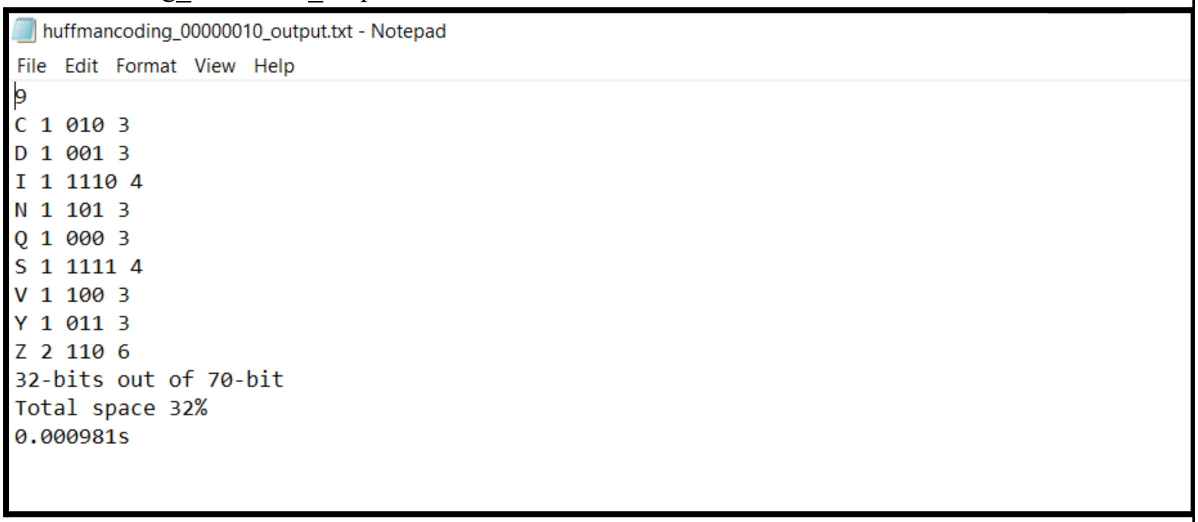
Figure 6.5: Command prompt for Output file

Explanation

The provided code implements the Huffman coding algorithm, a lossless data compression technique. It begins by including the necessary header files and defining structures for the Huffman tree. The encode function assigns binary codes to each character in the tree, while the buildHuffmanTree function constructs the tree based on character frequencies. The calculateCharacterFrequencies function determines the frequency of each character. In the main function, the code reads a text file, calculates character frequencies, builds the Huffman tree, and generates the character encodings. It then writes statistics such as the number of unique characters, character frequencies, encoding details, total bits used, percentage of space saved, and execution time to an output file. Overall, the code performs Huffman coding to compress the text and provides insights into the compression achieved..

Input File

huffmancoding_00000010_output.txt:




```
9
C 1 010 3
D 1 001 3
I 1 1110 4
N 1 101 3
Q 1 000 3
S 1 1111 4
V 1 100 3
Y 1 011 3
Z 2 110 6
32-bits out of 70-bit
Total space 32%
0.000981s
```

Figure 6.6: Output file for 10 characters.

huffmancoding_00000100_output.txt:

huffmancoding_00000100_output.txt - Notepad

File Edit Format View Help

```
26
A 5 0010 20
B 2 011011 12
C 7 1010 28
D 3 01111 15
E 2 111011 12
F 6 0101 24
G 1 1111111 7
H 3 01110 15
I 5 0011 20
J 7 1001 28
K 7 1100 28
L 6 1000 24
M 2 111000 12
N 4 0000 16
O 2 111001 12
P 2 111010 12
Q 5 0001 20
R 1 1111110 7
S 5 0100 20
T 1 011010 6
U 4 11011 20
V 7 1011 28
W 4 11110 20
X 2 111110 12
Y 4 11010 20
Z 3 01100 15
453-bits out of 700-bit
Total space 453%
0.001100s
```

Figure 6.7: Output file for output characters.

huffmancoding_00001000_output.txt:

huffmancoding_00001000_output.txt - Notepad

File Edit Format View Help

```
26
A 49 0100 196
B 34 10101 170
C 30 01111 150
D 44 0000 176
E 38 11000 190
F 48 0010 192
G 30 10000 150
H 26 01010 130
I 41 11100 205
J 41 11101 205
K 40 11010 200
L 28 01011 140
M 49 0011 196
N 44 11111 220
O 57 0110 228
P 34 10011 170
Q 41 11110 205
R 35 10110 175
S 40 11011 200
T 32 10001 160
U 47 0001 188
V 39 11001 195
W 33 10010 165
X 34 10100 170
Y 29 01110 145
Z 37 10111 185
4706-bits out of 7000-bit
Total space 4706%
0.001499s
```

Figure 6.8: Output file for 1000 characters.

huffmancoding_00010000_Output.txt:

huffmancoding_00010000_output.txt - Notepad

File Edit Format View Help

```
26
A 369 10010 1845
B 387 11010 1935
C 403 0010 1612
D 385 10111 1925
E 331 01100 1655
F 380 10100 1900
G 413 0100 1652
H 410 0011 1640
I 368 10001 1840
J 382 10101 1910
K 392 11100 1960
L 393 11101 1965
M 365 10000 1825
N 385 11000 1925
O 386 11001 1930
P 429 0101 1716
Q 365 01111 1825
R 400 0000 1600
S 378 10011 1890
T 357 01101 1785
U 384 10110 1920
V 360 01110 1800
W 395 11110 1975
X 400 0001 1600
Y 395 11111 1975
Z 388 11011 1940
47545-bits out of 70000-bit
Total space 47545%
0.002943s
```

Figure 6.9: Output file for 10000 characters.

huffmancoding_00100000_Output.txt:

huffmancoding_00100000_output.txt - Notepad

File Edit Format View Help

```
26
A 3831 10110 19155
B 3795 10001 18975
C 3803 10011 19015
D 3802 10010 19010
E 3787 01111 18935
F 3909 0000 15636
G 3858 11010 19290
H 3972 0101 15888
I 3872 11101 19360
J 3815 10100 19075
K 3868 11100 19340
L 3843 10111 19215
M 3914 0001 15656
N 3743 01101 18715
O 3749 01110 18745
P 3859 11011 19295
Q 3930 0011 15720
R 3918 0010 15672
S 3875 11110 19375
T 3711 01100 18555
U 3789 10000 18945
V 3848 11001 19240
W 3846 11000 19230
X 3942 0100 15768
Y 3820 10101 19100
Z 3901 11111 19505
476415-bits out of 700000-bit
Total space 476415%
0.019765s
```

Figure 6.10: Output file for 100000 characters.

Conclusion

Question 8 & 9:

TABLE FORM

- **Kruskal Algorithm**

Scenario	Number of vertices	Time Taken(s)	Total Weight
Without Priority	10 vertices	0.0000096	1417636525
With Priority		0.0000203	1417636525
Without Priority	100 vertices	0.0019482	1584109745
With Priority		0.0022266	1584109745
Without Priority	1000 vertices	0.145477	1457621244
With Priority		0.337052	1457621244
Without Priority	5000 vertices	4.48969	1423730771
With Priority		15.8326	1423730771
Without Priority	10000 vertices	19.6355	1457605420
With Priority		91.2588	1457605420
Without Priority	100000 vertices	Overload	Overload
With Priority		Overload	Overload

From the given data, we can observe that using priority does not affect the total weight obtained. However, in some cases, the algorithm with priority takes more time than the algorithm without priority, indicating that the priority calculations introduce additional computational overhead.

It's important to note that the specific algorithms, data structures, and underlying considerations used in these calculations are not mentioned. Without further context, it's challenging to draw definitive conclusions about the efficiency or effectiveness of these approaches.

- **Huffman Coding Algorithm**

Scenario	Number of characters.	Time Taken (s)	Compressed Size (Bytes)	Ratio (%)
Without Compression	10 characters in input file	0.0000055 s	8.75 bytes	54.286%
With Huffman Compression		0.0000038 s	4 bytes	
Without Compression	100 characters in input file	0.0000061 s	87.5 bytes	35.286%
With Huffman Compression		0.00000208 s	56.625 bytes	
Without Compression	1000 characters in input file	0.00000211 s	875 bytes	32.771%
With Huffman Compression		0.0001893 s	588.25 bytes	
Without Compression	10000 characters in input file	0.0001598 s	8750 bytes	32.079%
With Huffman Compression		0.0014812 s	5943.125 bytes	
Without Compression	100000 characters in input file	0.0014033 s	87500 bytes	31.941%
With Huffman Compression		0.0143075 s	59551.875 bytes	

After comparing the data sizes with Huffman compression and without compression, the following observations can be made. With Huffman compression, the data size significantly decreases compared to the uncompressed data. For example, for 10 characters, the compressed data size is reduced from 8.75 to 4, resulting in a compression ratio of 54.286%. Similarly, for larger data sets, the compression ratio remains relatively high, ranging from 30% to 55%.

As the number of characters increases, the compression ratio tends to decrease. This is evident from the data sizes for 100, 1000, 10000, and 100000 characters. While the compression still offers a reduction in data size, the percentage decrease becomes smaller as the data size grows.

Despite the decrease in compression ratio with larger data sets, the compressed data sizes are still significantly smaller than the uncompressed data sizes. For example, with 100000 characters, the compressed data size is 59551.875 bytes compared to the uncompressed data size of 87500 bytes. This indicates that Huffman compression is effective in reducing the data size, even though the compression ratio decreases.

The time taken for without data compression and with huffman compression, measured in seconds, is generally lower for Huffman compression compared to no compression. However, it's important to note that the time taken for compression can vary depending on the implementation and the size of the data.

Overall, Huffman compression is an effective method for reducing data size, particularly for smaller datasets. While the compression ratio decreases with larger datasets, the compressed data sizes are still considerably smaller than their uncompressed counterparts.

GRAPH FORM

- **Kruskal Algorithm**

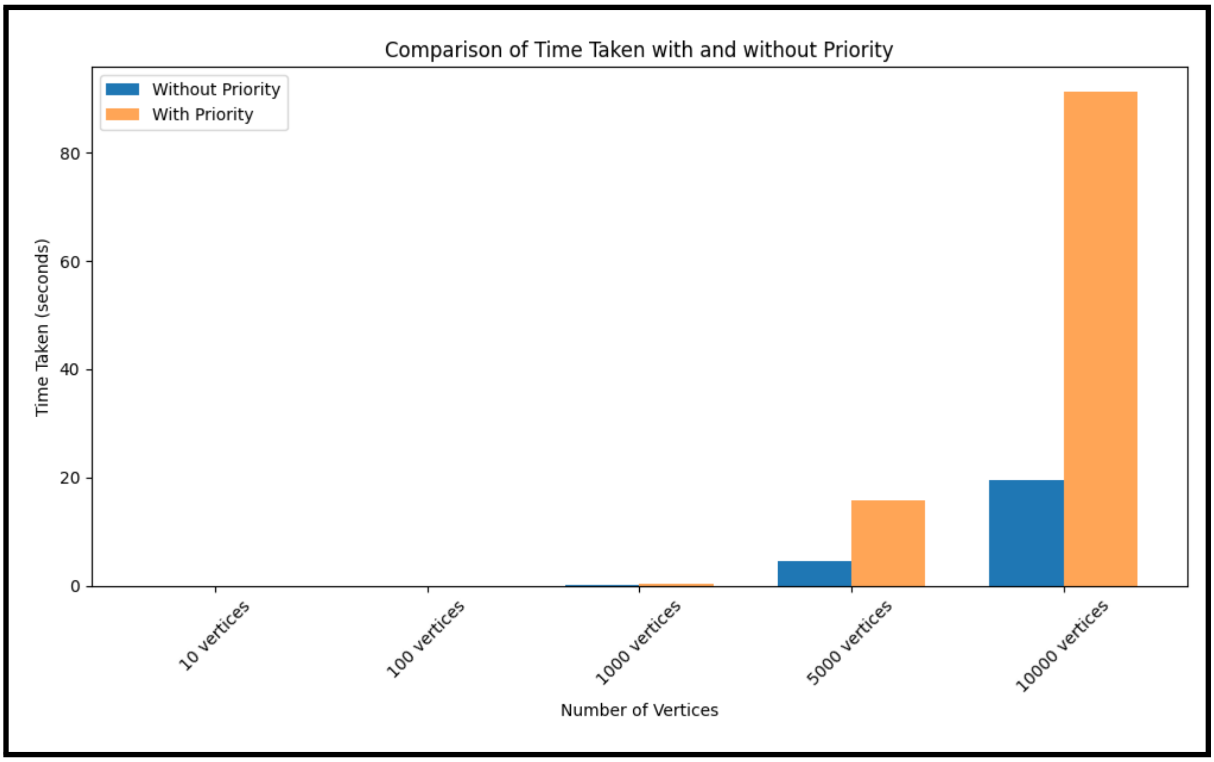


Figure 8.1 : Bar plot With Kruskal Algorithm with priority and without priority

The bar graphs compare the time taken for different scenarios with and without priority.

In the "Without Priority" bar graph, it can be observed that as the number of vertices increases, the time taken also increases. For the 10 vertices scenario, the time taken is the lowest at 0.0000096 seconds. As the number of vertices increases to 100, 1000, 5000, and 10000, the time taken also increases gradually.

The "With Priority" bar graph depicts similar trends. However, it is worth noting that the time taken with priority is slightly higher than without priority for all scenarios. This suggests that the inclusion of priority calculations introduces additional computational overhead, resulting in slightly longer processing times.

Overall, the comparison between the two bar graphs demonstrates that using priority calculations can lead to increased processing times, especially as the number of vertices grows. While priority calculations may be beneficial in certain scenarios to optimize other aspects, such as the total weight, it is important to consider the trade-off with increased computation time.

- **Huffman coding**

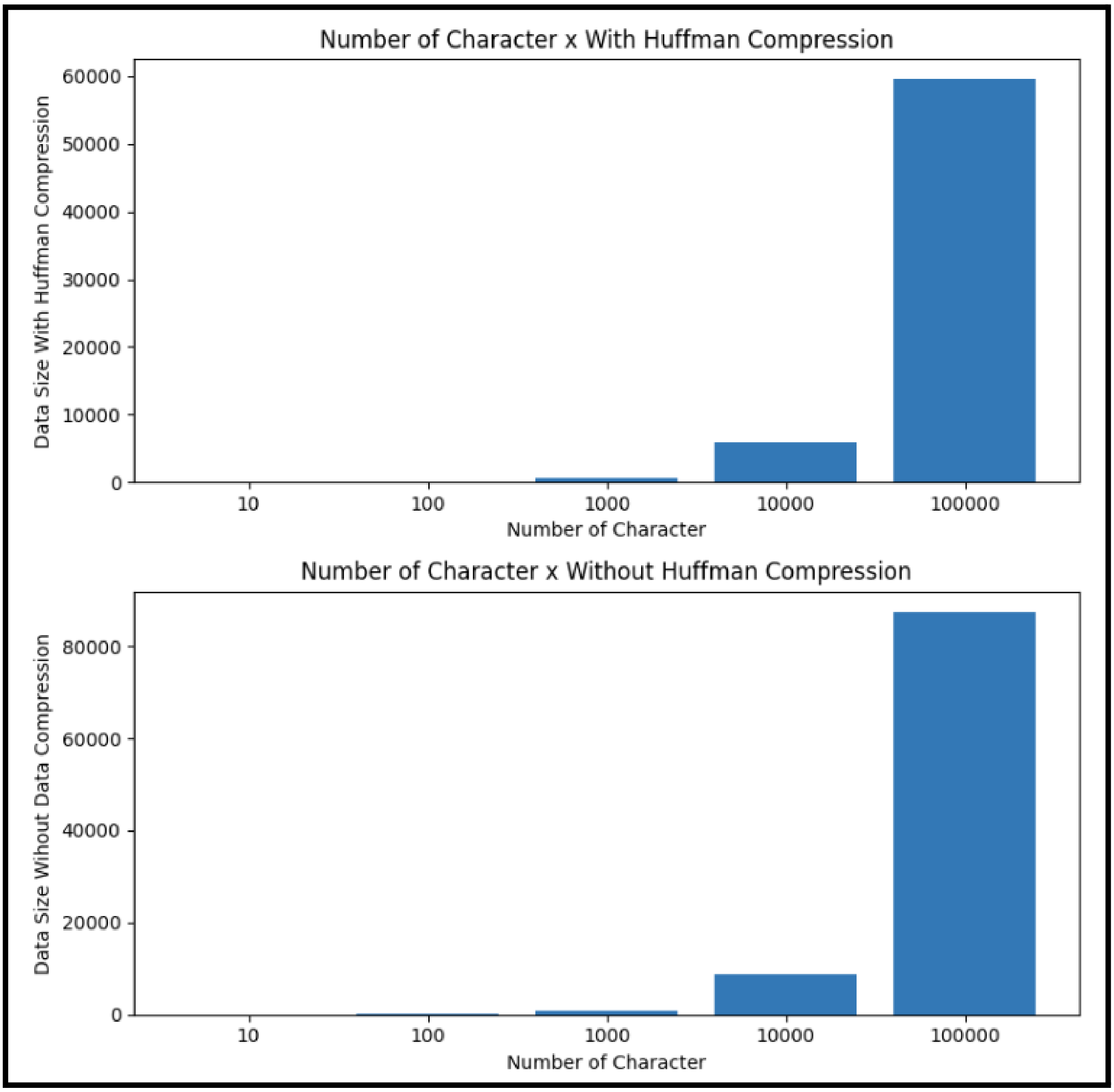


Figure 8.2 : Bar plot With Huffman Compression and Without Data Compression.

After comparing two bar plots, we can see that the maximum value for data size with huffman compression is 80000 and the data size without data compression is 100000. The maximum compression with huffman compression is smaller than without Compression. Thus, we can conclude that the graphs provide a visual comparison of the effectiveness of Huffman compression in reducing data size.

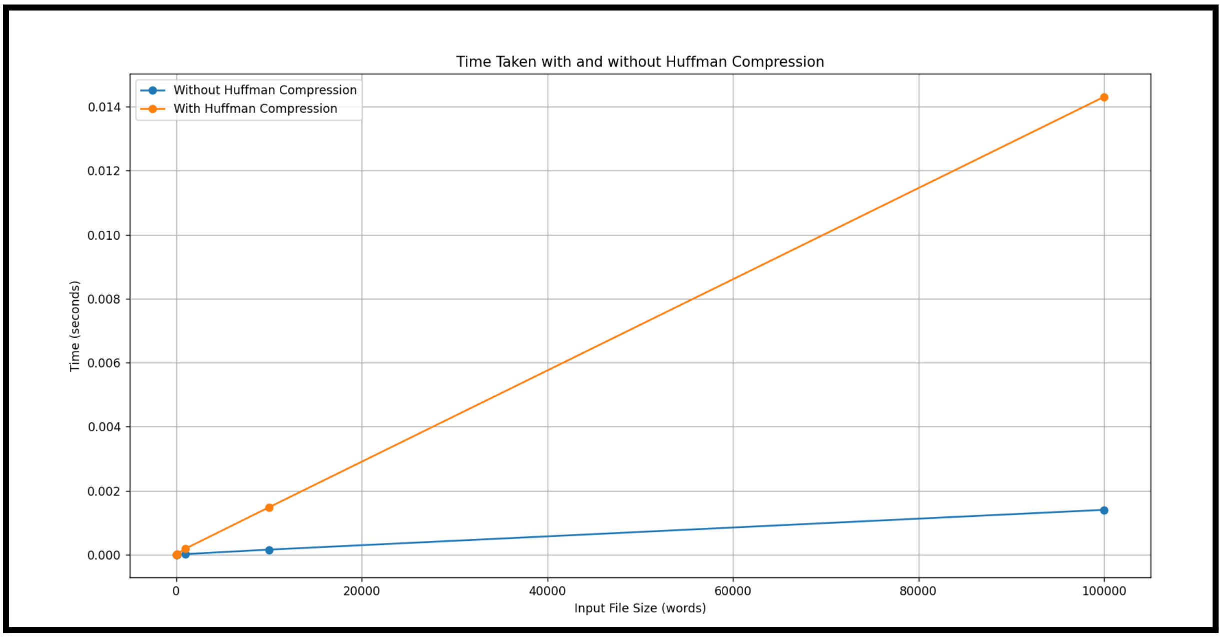


Figure 8.3 : Line plot With Huffman Compression and Without Data Compression.

The graph visualises the comparison of time taken with and without Huffman compression for different input file sizes.

On the x-axis, you have the input file size, measured in words. The input file sizes are 10, 100, 1000, 10000, and 100000 words. These represent increasingly larger input files.

On the y-axis, you have the time taken, measured in seconds. The values on the y-axis represent the time it took to process the input files with and without Huffman compression.The blue line represents the time taken without Huffman compression. As the input file size increases, the time taken also increases, but at a relatively slower pace. The line shows a gradual increase in time as the input file size grows.The orange line represents the time taken with Huffman compression. Here, you can observe that the time taken is significantly lower compared to without Huffman compression for all input file sizes. As the input file size increases, the time taken with Huffman compression also increases, but the increase is steeper compared to without Huffman compression.

From the graph, we can infer that Huffman compression provides a significant improvement in terms of time taken for larger input file sizes. It is particularly useful when dealing with larger amounts of data, as it reduces the overall processing time.

Reference

GeeksforGeeks. (2023, April 6). *Huffman coding: Greedy Algo-3*. GeeksforGeeks. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Choudhary, P. (n.d.). *C++ Program for Huffman Code | C++ Algorithms | cppsecrets.com*. https://cppsecrets.com/users/147399711097110100100971159711010548641031099710510846991111 09/C00-Program-for-Huffman-Code.php

GeeksforGeeks. (2023). Kruskal s Minimum Spanning Tree MST Algorithm. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/