Semester 04

Design and Analysis of Algorithm(CS2271)

Assignment 02:-

Group Members:-

1. Anish Banerjee (2021CSB001)
2. Joyabrata Acharya(2021CSB011)
3. Sk Fardeen Hossain(2021CSB023)

Source Codes and Documentations are available at Github

Indian Institute of Engineering Science and Technology, Shibpur

SL. NO.	TOPIC	SUBTOPICS
1	Polygon Triangulation	Preparing dataset of convex polygons
		Brute Force Approach
		Dynamic Programming Approach
		Greedy Approach
		Checking correctness of greedy Approach
2	Data Compression Strategies	Shannon-Fano encoding scheme
		Huffman Encoding Scheme
		Instantaneous decoding algorithm
		Huffman coding on a Large text document
		Examination of optimality of coding by comparing with other documents
3	Union Find on Large Real-Life Datasets	Realizing Disjoint set operations on large real-life datasets from SNAP and KONECT
		Implementing graph algorithms like connected-components and Minimum Spanning Tree using Union-Find Operations

Polygon Triangulation Problem

- Polygon triangulation problem is a famous computational geometry problem that tells us to find the minimum cost of triangulation.
- Triangulation refers to process of breaking the polygon to a number of triangles by drawing diagonals between non-adjacent vertices such that no two diagonals intersect.
- Cost refers to the cost of triangulation that is the sum of perimeters of all the triangles formed after triangulation has been done.
- For more we can refer here

Dataset Preparation

- Here we will be constructing convex polygons and regular polygons(which is also convex) by using concepts of vector algebra.
- A regular polygon of n vertices can easily be constructed using the fact that all the vertices are equidistant on a circle of given radius; giving the hint that they lie on the De-Moivre's Circle.

```
void generate(Vertex *arr, int n, int radius)
{
  for (int i = 0; i < n; i++)
  {
    arr[i].x = X0 + radius * cos((2 * PI * i) / n);
    arr[i].y = Y0 + radius * sin((2 * PI * i) / n);
  }
}
```

:- Code Snippet for generating regular polygons

Convex Polygon Generation

- A convex polygon can also be generated on the fact that while taking consecutive cross products of two adjacent sides of the polygon; the sign of the cross-product for all the pair of sides must remain same. This distinguishes a regular polygon from a convex one.
- The source code can be found in the repo, and we have used the above fact in our algorithm by iteratively checking for convexity while generating the vertices for our polygon.

Brute Force Approach

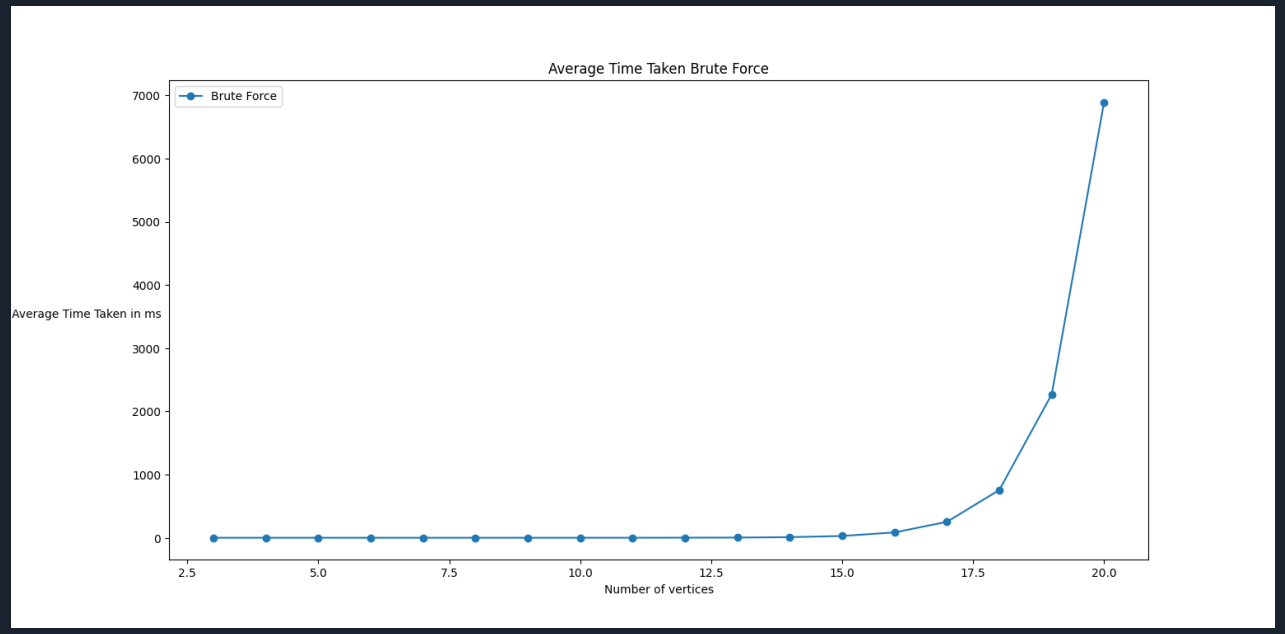
- We know that a convex polygon of n vertices can be triangulated into n-2 triangles, our job is to find the most optimal(minimum perimeter) combination of the possible n-2 triangles.
- The naïve approach to solving this problem is to look at all the possible cases, i.e., dividing the polygon into 2 sub-polygons and 1 triangle and then recursively adding the cost of the triangle to the cost of the 2 sub-polygons. The terminating condition for the recursion will be when there are less than equal to 2 vertices left in which case we will just return 0.

```
double minTriangulationCost(Vertex *points, int i, int j)
{
  // Base Case
  if (j <= i + 1)
  {
    return 0;
  }
  double ans = 1e7;
  for (int k = i + 1; k < j; k++)
  {
    ans = min(ans, (minTriangulationCost(points, i, k) + minTriangulationCost(points, k, j) + cost(points, i, k, j)));
  }
  return ans;
}
```

Code Snippet for brute force approach

Observation and Analysis of Brute Force Approach

- We have implemented the naïve approach on a dataset of regular convex whose vertices range from 3 to 21 and each vertices has around 50 rounds for a clear average case analysis.
- Theoretically it seems that the time complexity is exponential and it seems to be of the order of the catalan number, thus the time complexity in the worst case and average case will be 4^n/(n^1.5).



As we can see from the figure aside, the time taken for the naïve approach to compute the min. Triangulation cost is exponentially rising with the number of vertices.

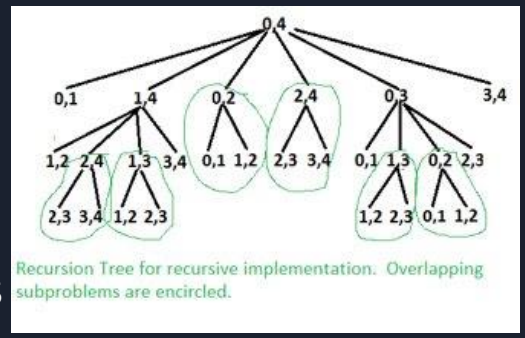
The question comes that can we optimize this approach to reach a polynomic time complexity

Dynamic Programming Approach

- The dynamic programming approach uses the fact that the recursive solution posed by the naïve approach had an optimal substructure and overlapping subproblems. This can be justified from the fact that when we are checking for triangulation possibilities the shortest perimeter triangles will always come in our answer that is they will always be found being overlapped with different branches in the recursion tree.
- So the natural implication will be to store the minimum value result in a table for indices say I to J and whenever we need to find the cost for triangulation of indices from I to J we can easily look it up from the table thus preventing extra computation
- The table will be 2 dimensional as one dimension is for the start index and the other dimension is for the end index. So the answer will be dp[0][n-1]; considering n vertices.

```
double minTriangulationCostDP(Vertex *points, int n)
{
  double Max = 1e7;
  if (n <= 2)
  {
    return 0;
  }
  /* table[i][j] will store the min triangulation cost for vertices from point
    i to j. Final ans will table[0][n-1]
  */
  for (int m = 0; m < n; m++)
  {
    for (int i = 0, j = m; i < n; i++, j++)
    {
      if (j <= i + 1)
      {
        dp[i][j] = 0.0;
      }
      else
      {
        dp[i][j] = Max;
        for (int k = i + 1; k < j; k++)
        {
          double ans = dp[i][k] + dp[k][j] + cost(points, i, j, k);
          dp[i][j] = min(dp[i][j], ans);
        }
      }
    }
  }
  return dp[0][n - 1];
}
```

Code snippet for DP approach

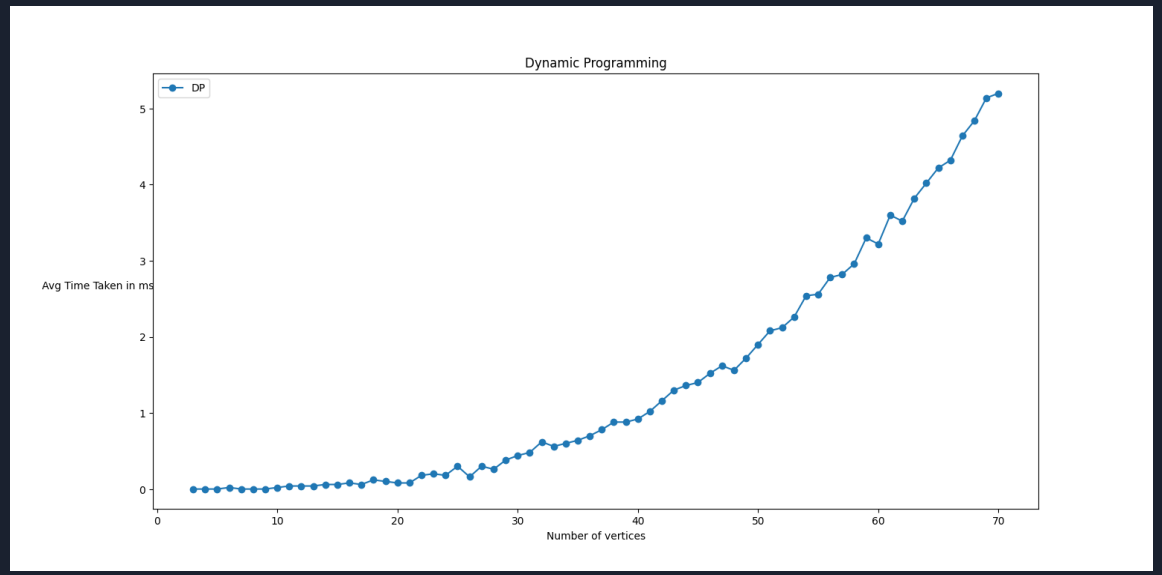


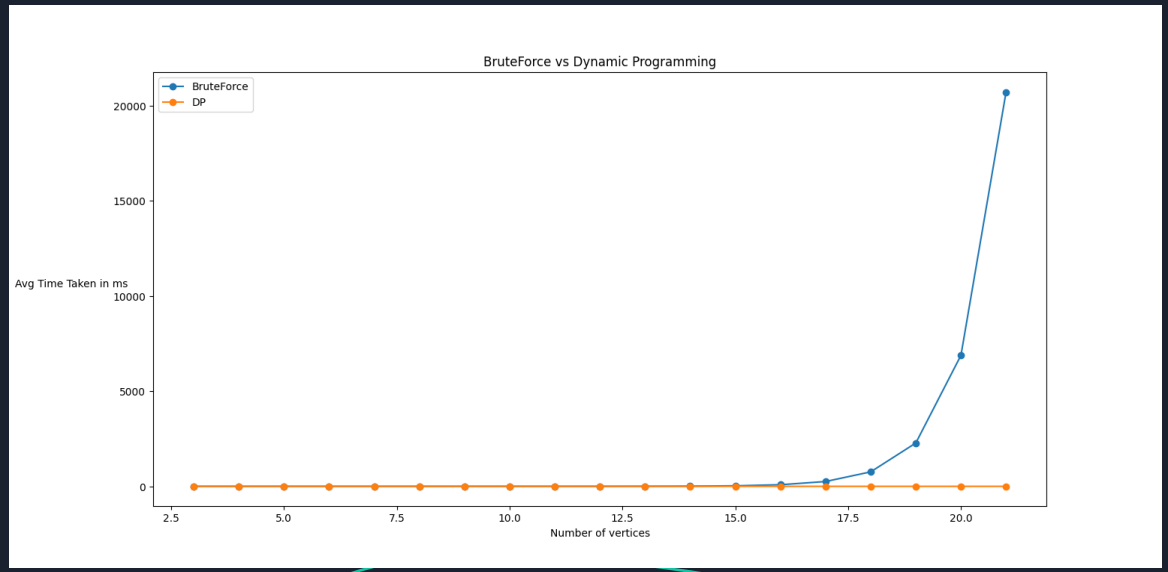
Recursion Tree for recursive implementation. Overlapping subproblems are encircled.

From the code aside it is pretty obvious that we have to loop for all vertices(2 loops) and then check all possible intermediate vertices(1 loop), we have 3 loops in our functions so the worst case time Complexity will be O(N^3), N being the number of vertices

Observation and Analysis of DP Approach

- From the snippet we have seen in the previous slide, the dynamic programming approach leads to worst case time complexity of O(N^3) which significantly reduces it from a non-polynomic time –complexity provided by the naïve approach.





Greedy Strategy

- Another strategy to solve this problem is to greedily select non-intersecting diagonals in an ascending order of length by ear clipping the polygon at three adjacent vertices and adding them to our answer .
- The problem with this greedy approach is that it does not satisfy the exchange property when this problem is mapped to a greedy matroid, because it does not guarantee the optimality of the solution that adding shorter perimeter triangles will not lead to larger perimeter triangles since we are already narrowing the search space by the keyword 'non-intersecting' which must be followed.
- So it is expected that the greedy solution will not give optimal results than the DP one but it is helpful in situations which demands time and there is less significance to minute deviation from the optimal result.

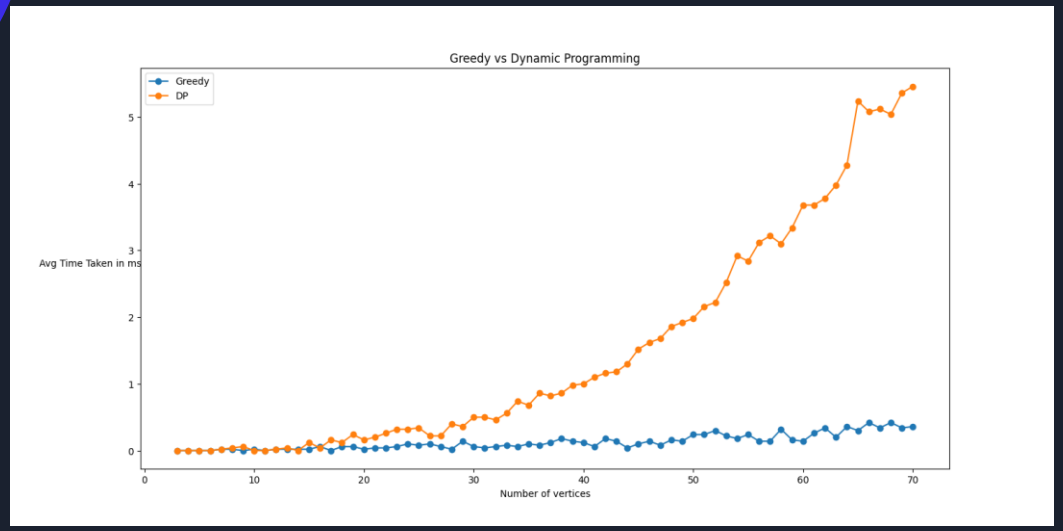
Code Snippet

```
// Ear Clipping Algorithm
double minTriangulationCostGreedy(Vertex *points, int *visited, int n)
{
  if (n == 3)
  {
    return trigCost(points[visited[0]], points[visited[1]], points[visited[2]]);
  }
  int index;
  int newVisited[n - 1];
  double min = 1e7;
  double cost;

  for (int i = 1; i < n - 1; i++)
  {
    cost = trigCost(points[visited[i - 1]], points[visited[i]], points[visited[i + 1]]);
    if (cost < min)
    {
      min = cost;
      index = visited[i];
    }
  }
  cost = trigCost(points[visited[n - 1]], points[visited[n - 2]], points[visited[0]]);
  if (cost < min)
  {
    min = cost;
    index = visited[n - 1];
  }
  cost = trigCost(points[visited[0]], points[visited[1]], points[visited[n - 1]]);
  if (cost < min)
  {
    min = cost;
    index = visited[0];
  }
  int m = 0;
  for (int i = 0; i < n; i++)
  {
    if (visited[i] != index)
    {
      newVisited[m++] = visited[i];
    }
  }
  return cost + minTriangulationCostGreedy(points, newVisited, n - 1);
}
```

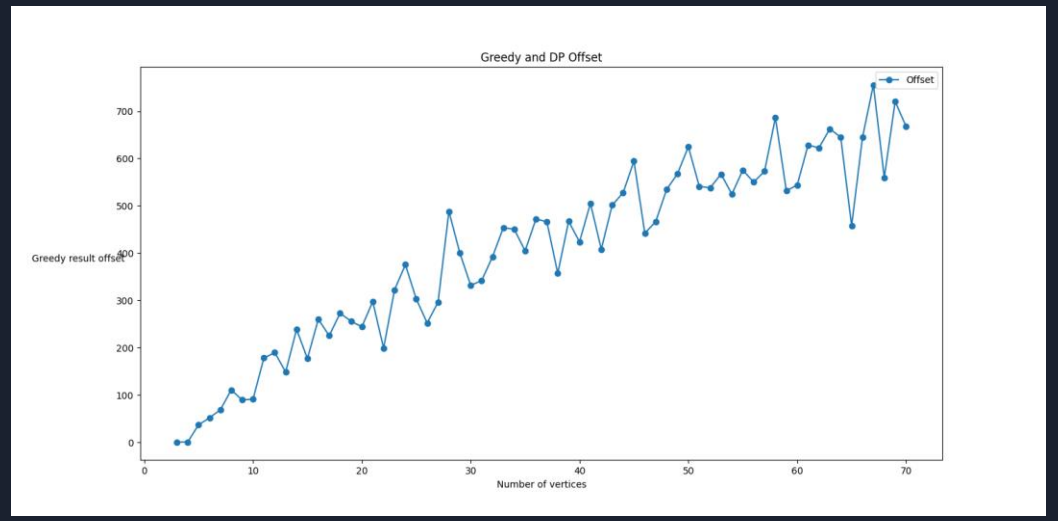
From the code snippet aside we can clearly see that we are clipping smaller perimeter triangles from the polygon and adding its perimeter to our answer. We maintain the non-intersection condition by maintaining a visited array whose size keeps on decreasing by 1 in the recursive cycle.

Each recursive cycle in the worst case performs O(n) work and there are n-2 recursions taking place(including the base case which takes O(1) time) the total time complexity of the greedy algorithm will be O(N^2) which is better than the DP but the result will vary due to lack of optimal structure in the greedy approach.

Observation



Greedy vs DP (Time Taken)



Offset of how much Greedy result is more than that of the optimal DP result.

We can clearly see that the DP and the Greedy results show mismatch which becomes significant when the number of vertices in the polygon is significantly high.

2. Data Compression Strategies: Shannon Fano and Huffman

- **Data compression** is a technique used to reduce the size of data by encoding it in a more efficient manner. Two very popular algorithms for data compression are Shannon-Fano and Huffman coding.
- **Shannon-Fano** coding is a prefix code where each symbol is assigned a binary code based on its probability of occurrence in the data.
- **Huffman coding** is also a prefix code where each symbol is assigned a variable length binary code based on its frequency of occurrence in the data.
- Both are lossless data compression techniques, meaning that there is no loss of data in data compression and decompression.

Background...

The problem of finding the prefix code with the minimum expected length is equivalent to finding the set of lengths $l_1, l_2, \ldots, l_m$ satisfying the Kraft inequality and whose expected length is $L = \Sigma l_i p_i$ less than the expected length of any other prefix code.

The mathematics yields us the following result: $H(X) \leq L < H(X) + 1$ where $H(X)$ here is the file entropy that gives a quantitative measure of the degree of randomness in the file.

($H(X)$ is calculated as $\Sigma\ p_i \log(1/p_i)$)

The above inequality shows that the average codeword length L for the prefix code must be greater than or equal to the entropy H of the source.

This relationship between entropy and codeword length is a fundamental result in information theory, and it shows that the optimal average codeword length for a prefix code is equal to the entropy of the source.

2(a) Encoding algorithm for Shannon-Fano

How it works:

Step 1:

SYMBOL	A	B	C	D	E
PROBABILITY OR FRQUENCY	0.22	0.28	0.15	0.30	0.05

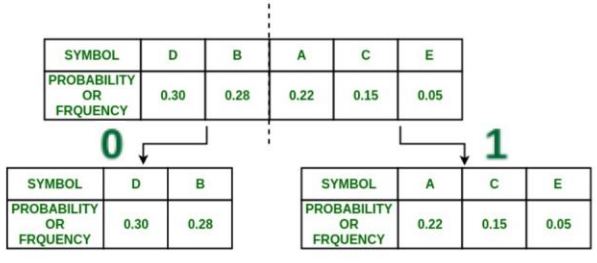
THE SYMBOLS AND THEIR PROBABILITY / FREQUENCY ARE TAKEN AS INPUTS.
(In case of Frequency, the values can be any number)

Step 2:

SYMBOL	D	B	A	C	E
PROBABILITY OR FRQUENCY	0.30	0.28	0.22	0.15	0.05

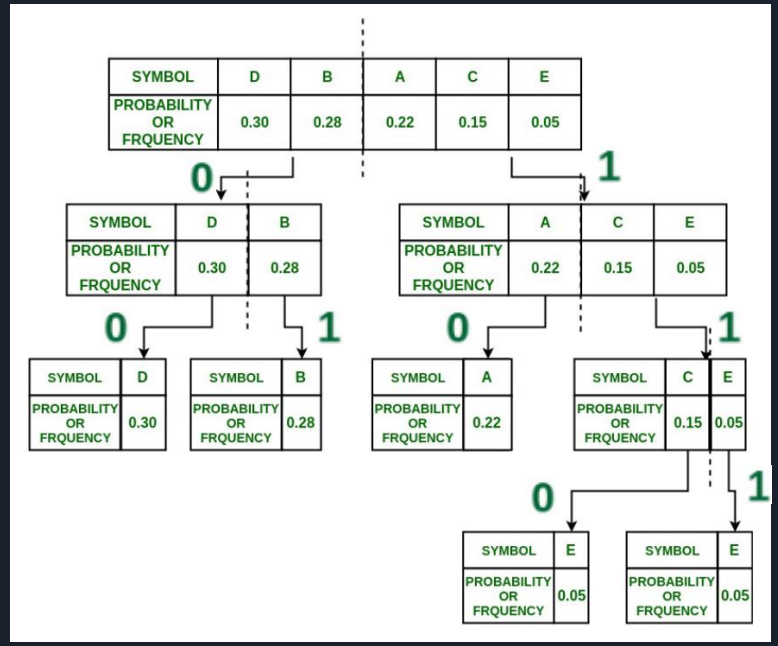
INPUTS ARE SORTED ACCORDING TO THEIR PROBABILITY / FREQUENCY
(Here they are sorted according to their probability)

Step 3:



THE SYMBOLS ARE DIVIDED INTO TWO SUCH THAT THE TOTAL PROBABILITY / FREQUENCY OF LEFT SIDE ALMOST SAME AS THAT OF RIGHT SIDE

Step 4:



Step 5:

SYMBOL	A	B	C	D	E
PROBABILITY OR FRQUENCY	0.22	0.28	0.15	0.30	0.05
SHANNON-FANO CODE	00	01	10	110	111

THE SHANNON CODES OF THE SYMBOLS
(based on their traversal from the root node)

Code for assigning Shannon Codes:

```
typedef struct{
    char symbol;
    int freq;
    double prob;
    char code[40];
    int top;
} char_data;
```

```
void shannon_fano(char_data* char_freq, int
start,    int end)
{
    if (start >= end)    return;

    double total_prob = 0;
    for (int i = start; i <= end; i++)
        total_prob += char_freq[i].prob;

    double half_prob = total_prob / 2.0;
    double current_prob = 0;
    int split_index = -1

    if(end - start >= 2)
        for (int i = start; i <= end; i++)
            current_prob += char_freq[i].prob;
            if (current_prob >= half_prob){
                split_index = i-1;
                if(split_index < start)
                    split_index++;
                break;
            }
    else
        split_index = start;
```

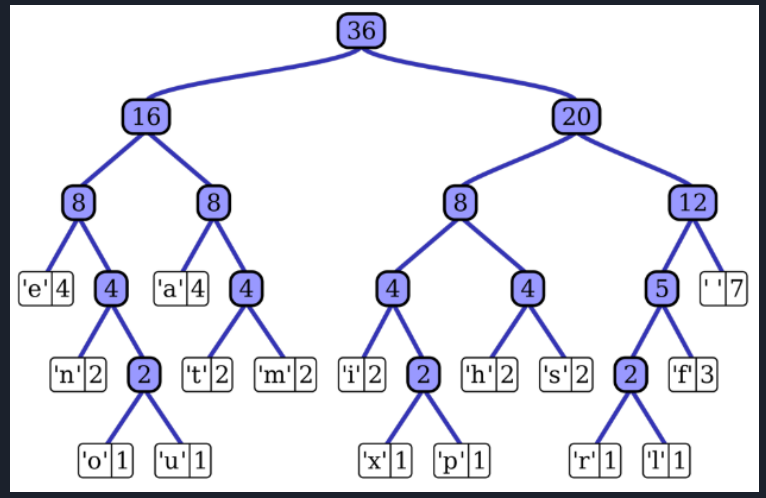
```
for (int i = start; i <= end; i++){
        int index_offset =
            (i <= split_index) ? 0 : split_index+1;
        char bit = (i <= split_index) ? '0' : '1';
        char_freq[i].code[++char_freq[i].top] = bit;
        char_freq[i].code[char_freq[i].top+1] = '\0';
    }

    shannon_fano(char_freq, start, split_index);
    shannon_fano(char_freq, split_index+1, end);
}
```

Procedure for Compression:

1. Assumption: A maximum of 256 ASCII characters can be present in the text file. So, we create an array of 256 elements to store the Shannon Fano codes. Index of each element corresponds to its ASCII value. This is our codebook for the file.
2. We open the document to be compressed, read character by character and corresponding to each character we visit the relevant array position in the codebook to get the Shannon Fano code.
3. Then we write the codeword for the character in a separate new file to store the encoded form of the text file.

Procedure for Decompression:

1. The codebook is used to create a Shannon Fano tree. This is basically a binary tree such that the higher probability characters reside at the upper portion of the tree.
2. During traversal of the encoded text file, we start from the root of the Shannon Fano tree and move left or right based on whether the current bit is 0 or 1. We continue this process until we reach a leaf node, at which point we output the corresponding character to the decoded text file and start again from the root of the tree.

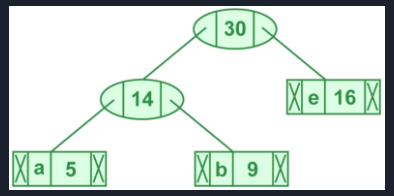


2(b) Huffman encoding using heap

How it works:

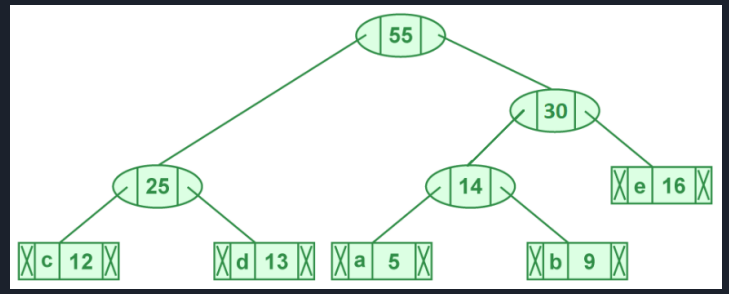
Step 1:

Step 2:

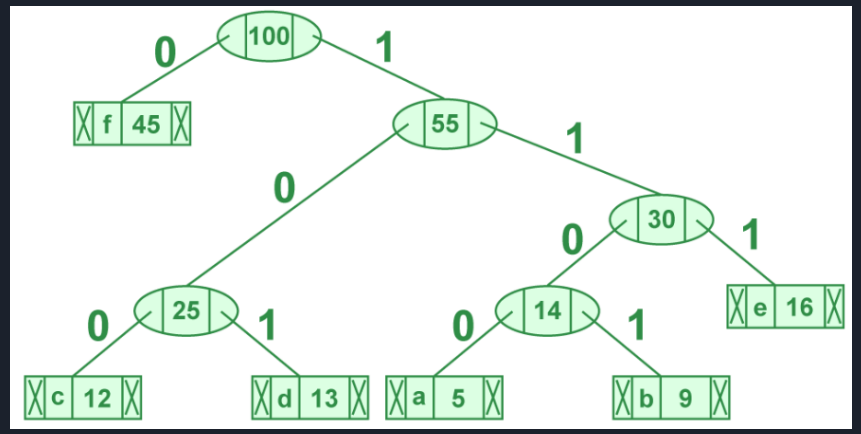


character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 3:



Step 4:



character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Code for assigning Huffman codes:

```
struct Node{
    char ch;
    int freq;
    string code;
    Node* left, * right;
    Node(char ch, int freq);
};
```

```
Node* buildHuffmanTree(unordered_map<char, int>& freq_map){
    Node* nodes[freq_map.size()];  // to store the nodes of the Huffman Tree
    int i = 0;
    for (auto& pair : freq_map)
        nodes[i++] = new Node(pair.first, pair.second);

    // Build the Huffman Tree using a heap
    while (i > 1){
        int min1 = 0, min2 = 1; //min1 for smallest frequency and min2 for second smallest
        if (nodes[min1]->freq > nodes[min2]->freq)
            swap(min1, min2);

        for (int j = 2; j < i; j++)
            if (nodes[j]->freq < nodes[min1]->freq){
                min2 = min1;    min1 = j;
            }
            else if (nodes[j]->freq < nodes[min2]->freq)
                min2 = j;

        // Create a new node with the sum of the frequencies of the two smallest nodes
        Node* parent = new Node('\0', nodes[min1]->freq + nodes[min2]->freq);
        parent->left = nodes[min1];
        parent->right = nodes[min2];
        // Remove the two smallest nodes from the array and add the new node
        nodes[min1] = parent;
        nodes[min2] = nodes[i - 1];
        i--;
    }
    return nodes[0]; // Return the root of the Huffman Tree
}
```

Compressing a File using Huffman Codes:

Count the frequency of each character in the input file and store it in a frequency map

Build the Huffman Tree using the frequency map

Traverse the Huffman Tree and store the Huffman codes in a map

Encode the input file using the Huffman codes and write the encoded data to output file

```
void encodeFile(string original_file, string encoded_file)
{
    unordered_map<char, int> freq_map;
    char ch;
    ifstream infile(original_file);
    while (infile >> noskipws >> ch)
        freq_map[ch]++;
    infile.close();

    Node* root = buildHuffmanTree(freq_map);

    unordered_map<char, string> Codemap;
    traverse(root, "", Codemap);

    ofstream outfile(encoded_file);
    infile.open(original_file);
    while (infile >> noskipws >> ch)
        outfile << Codemap[ch];

    infile.close();
    outfile.close();
}
```

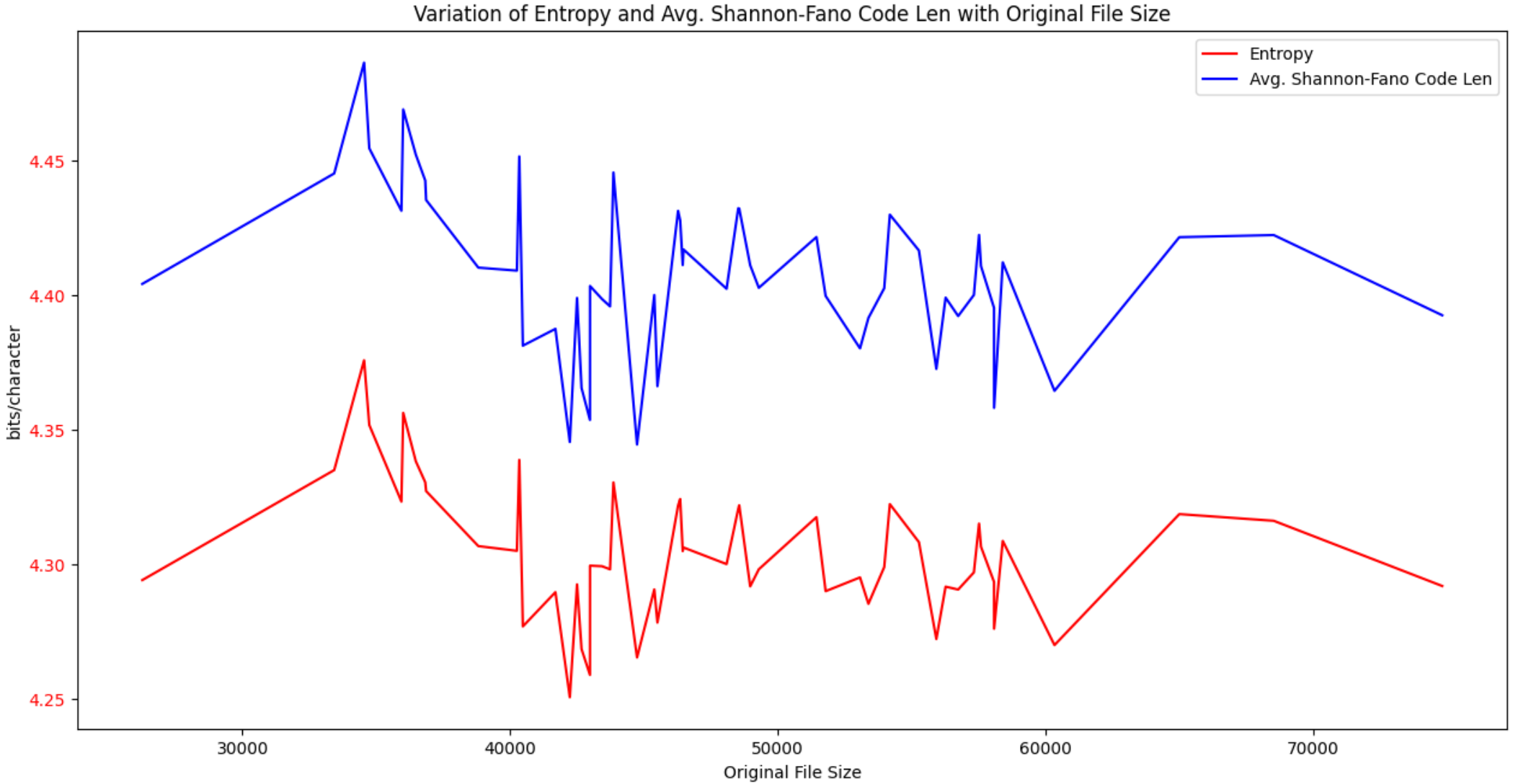
Decompressing the File:

```
// Traverse the Huffman tree to decode the input string
void decodeFile(Node* root, const string encoded_file, const string decoded_file)
{
    // Open the input and output files
    ifstream inFile(encoded_file, ios::binary);
    ofstream outFile(decoded_file, ios::binary);

    // Traverse the Huffman tree for each character in the input file
    Node* current = root;
    char c;
    while (inFile.get(c))
    {
        if (c == '0')
            current = current->left;
        else
            current = current->right;

        if (current->left == nullptr && current->right == nullptr)
        {
            outFile.put(current->ch);
            current = root;
        }
    }

    // Close the input and output files
    inFile.close();
    outFile.close();
}
```

Experimental Procedure :

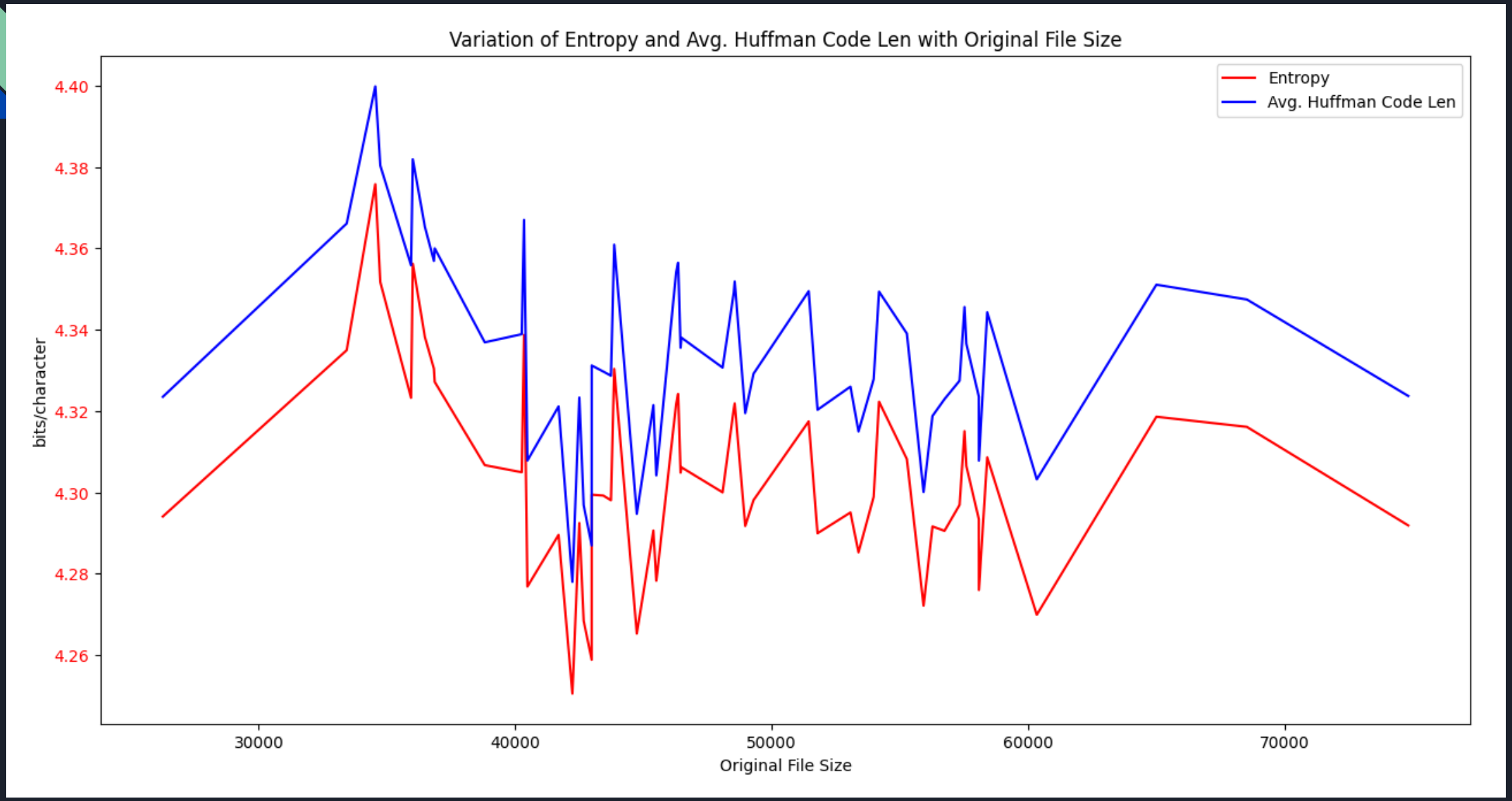
- A sample of 54 large text documents were taken up for data compression/ decompression experiment
- **Shannon Fano Experiment:**
  - Each of the files were individually compressed and compression ratios were plotted
  - To get a better understanding of the degree of compression, the average codeword length was plotted along with the theoretical entropy values of each of the files.
- **Huffman Experiment:**
  - **Experiment I:**
    - A cumulative frequency-distribution of all the characters in all the text files was taken and based on that, Huffman codes were assigned. This was then used to compress the file
    - Once again, the relevant plots were recorded
  - **Experiment II**
    - This time, instantaneous encoding-decoding were done using Huffman
  - **Experiment I vs II** were plotted to get an understanding of how Experiment I result deviates from the practical optimum Huffman compression
- **Huffman vs Shannon Fano** plots were done to observe which of them on an average gave better results for compression

Shannon Fano Experiment Result:

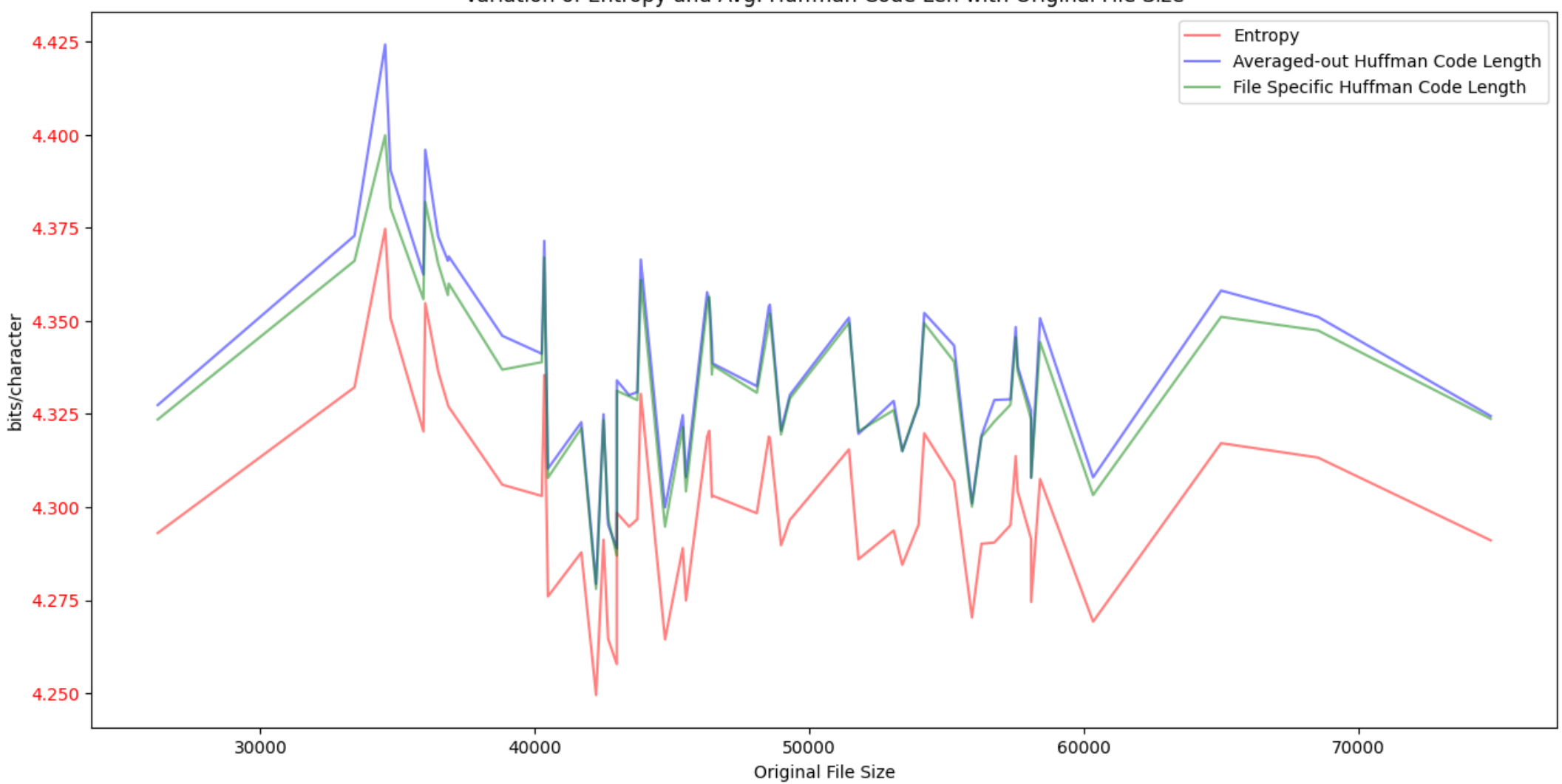


Huffman Experiment I (Cumulative Frequency):

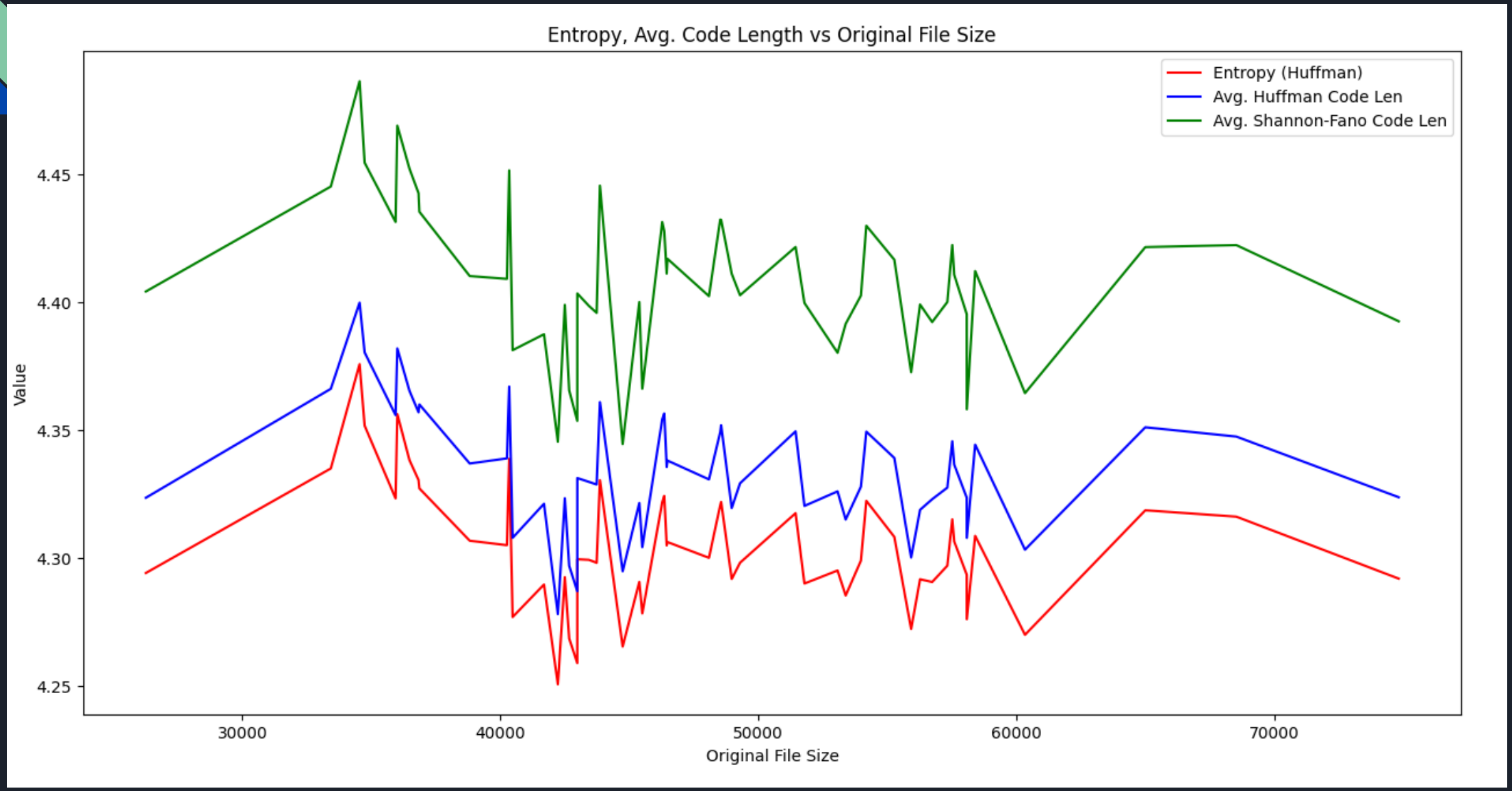
Huffman Experiment II (Instantaneous):



Huffman Experiment I vs II:



Shannon Fano vs Huffman (Instantaneous)



Observations :

1. **Compression Ratio:** Huffman coding generally achieves better compression ratios compared to Shannon-Fano. This is because Huffman coding generates the most efficient code words for each symbol based on their frequency of occurrence using optimal merge pattern, whereas Shannon-Fano uses a divide-and-conquer approach that may not always result in the optimal code words.
2. **Optimal substructure:**
    a. In **Huffman coding**, the optimal prefix codes for a given set of symbols can be obtained by combining the optimal prefix codes of its subsets of symbols. So Huffman algorithm is able to find the optimal prefix code for a set of symbols by solving smaller subproblems.
    b. **Shannon-Fano coding** does not exhibit optimal substructure because it uses a divide-and-conquer approach that may not result in the optimal code words for each symbol.
3. **Greedy Choice Property:**
    a. At each step, Huffman coding selects the two symbols with the lowest probabilities and merges them into a single node. This greedy choice ensures that the optimal prefix codes are obtained for the entire set of symbols.
    b. At each step, Shannon-Fano coding splits the remaining symbols into two groups based on their probabilities, and assigns a 0 or 1 bit to each group. This greedy choice may not always result in the optimal prefix codes for the entire set of symbols.

PART - III

- 3. We need to scale up the things - from toy problems to real life problems. For this, you need to study some large datasets provided by reputed Universities, like
- **(a) SNAP -** Stanford Network Analysis Project - datasets collected by Stanford University.
- **(b) KONECT -** Koblenz Network Collection - datasets compiled by Koblenz University, Deutschland (Germany).

- First study and feel the vastness of these networks from the dynamic and disjoint set operations point of view. Then try to implement graph algorithms like **Connected Components** and **Minimum Spanning Tree** using appropriate data structure for these datasets.

THE CONCEPTS

- Minimum Spanning Tree (MST) is a concept in graph theory that refers to a subset of edges of a connected, weighted graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. In other words, an MST is a tree (i.e., a connected graph without cycles) that spans (i.e., covers) all the vertices of a graph with the minimum possible total weight.
- Connected components refer to the subgraphs of a graph that are connected (i.e., there is a path between any two vertices) and cannot be further divided into smaller connected subgraphs. In other words, a connected component is a maximal subgraph in which every vertex is reachable from any other vertex.

SAMPLE DATA

OUTPUTS

MEMBER

```
1 2
2 2
3 1
4 3
5 1
6 3
```

CONNECTIVITY FILE

```
1 2
2 5
4 6
```

CONNECTED COMPONENTS O/P

```
Number of connected components: 3
Connected components:
1 2 5
3
4 6
```

MINIMUM SPANNING TREE (MST) O/P

```
Minimum Spanning Tree:
( 1 , 2 )
( 3 , 1 )
( 4 , 3 )
( 5 , 1 )
( 6 , 3 )
```

^ TESTING THE ALGORITHMS ON A SAMPLE TOY DATASET ^

Implementation Details

- We have implemented the disjoint set union data-structure using linked-list and tree-based approaches.
- We have tested the algorithm on:-

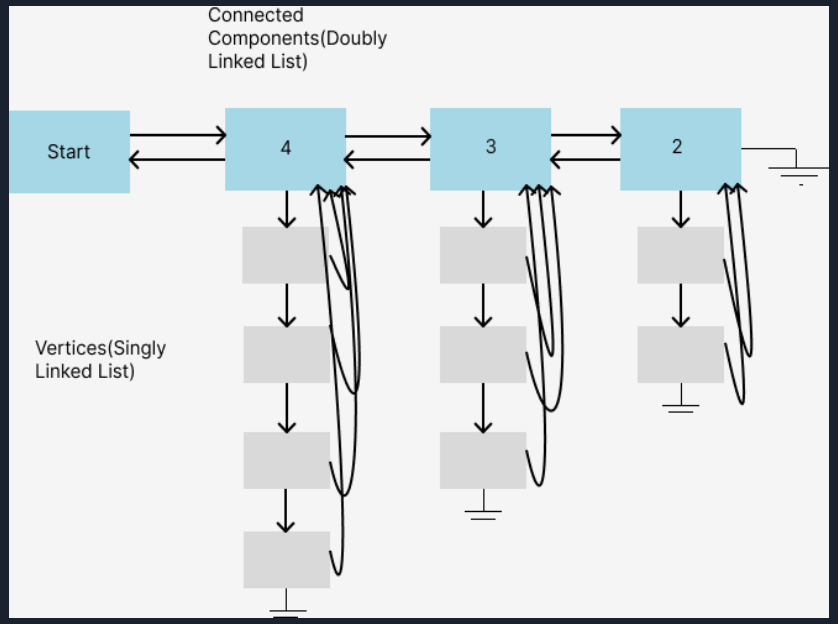
1. SNAP (Email-EU-Core-Network)
2. SNAP (COM-YouTube)
3. SNAP (EGO-Facebook)
4. KONECT (Full USA Road-Map)

Dataset Generic Details

- All the datasets used have a similar format (all .txt)
- Dataset had commented lines in the start which had number of vertices and edges of the dataset
- After that every line was a set of two numbers, which defined an edge. Using these two properties it was easy to derive a way to extract data from the dataset.

Linked List Based Implementation

- Here we used Two Linked Lists, One for dynamically managing number of Connected Components of the graph (also called Representative(Set) Element), and other to maintain the list of vertices inside that connected component.
- The Connected Component list is doubly linked while the vertices list in singly linked.
- Every Representative Node also has a length parameter(for union operations), which stores the length of vertex linked list inside it.
- Every vertex also has a representative pointer which points to its Connected Component Node
- The amortized time complexity in this implementation is O(log N), N being the number of edges in the dataset.



Code Snippet

```
// Initialisation of all nodes to a default value; creation of new set

void MAKE_SET(Long Long index, ConComp *start)
{
  ConComp *temp = start->next;

  ConComp *ans = new ConComp;
  ans->length = 1;
  ans->next = temp;
  start->next = ans;
  ans->prev = start; // doubly linked list

  if (temp != NULL)
  {
    temp->prev = ans;
  }

  ans->head = new node;
  ans->tail = ans->head; // initially head will be tail
  ans->head->data = index;
  ans->head->parent = ans;
  ans->head->next = NULL;
}
```

```
// Find the representative element
ConComp *FIND_SET(Long Long val, ConComp *start)
{
  ConComp *temp = start->next;
  while (temp != NULL)
  {
    node *node_temp = temp->head;
    while (node_temp != NULL)
    {
      if (node_temp->data == val)
      {
        return node_temp->parent;
      }
      node_temp = node_temp->next;
    }
    temp = temp->next;
  }
  return NULL;
}
```

```
void LINK(ConComp *x, ConComp *y)
{
  if (x != y)
  {
    ConComp *max, *min;
    if (x->length > y->length)
    {
      max = x;
      min = y;
    }
    else
    {
      max = y;
      min = x;
    }
    max->length += min->length;
    node *point_diff = min->head;
    max->tail->next = min->head;
    max->tail = min->head;

    while (point_diff != NULL)
    {
      point_diff->parent = max;
      point_diff = point_diff->next;
    }
    if (min->prev != NULL)
    {
      min->prev->next = min->next;
    }
    if (min->next != NULL)
    {
      min->next->prev = min->prev;
    }
    delete min;
  }
}

// Union of two sets

void UNION(edge e, ConComp *start)
{
  if (e.source != e.dest)
  {
    LINK(FIND_SET(e.source, start), FIND_SET(e.dest, start));
  }
}
```

```
struct ConComp // Structure for the connected components
{
  ConComp *next;
  ConComp *prev;
  node *head;
  node *tail;
  Long Long length;
};

struct node
{
  ConComp *parent;
  Long Long data;
  node *next;
};

struct edge
{
  Long Long source;
  Long Long dest;
};
```

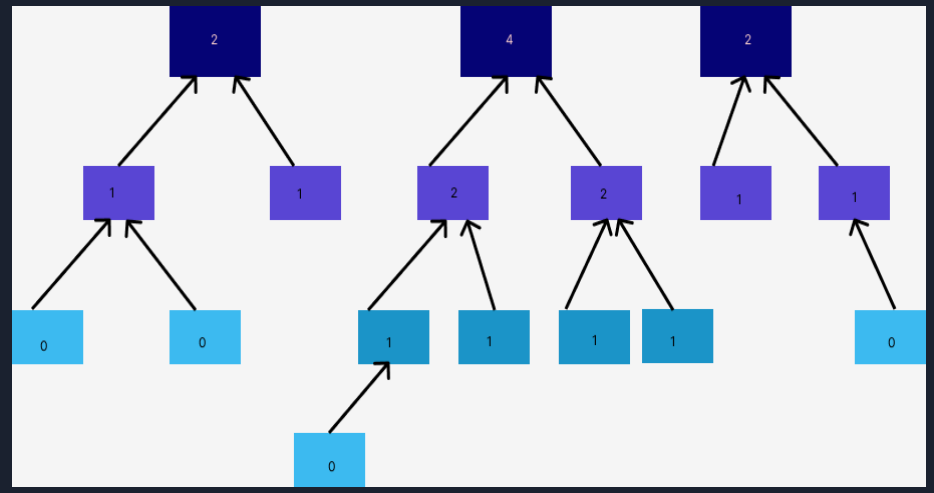
Tree Based Implementation

- Here we have only vertex as nodes, which have two extra data other than its value :-

1. Rank

2. Parent Pointer

- Rank of a node signifies how big the subtree with the node as root is. Parent pointer of a node points to one of the nodes which are the parents of the node. Preferable to the node with highest node in the tree (the main root)
- Expected Amortized Time Complexity in this Implementation is:- O(log*N); N being the number of edges
- The above implementation can also be realized using vectors(CPP)



Code Snippet

```
// Initialisation of all nodes to a default value; creation of new set

void MAKE_SET(node *p)
{
  p->parent = p; // initially a node is its own parent
  p->rank = 0;   // initially rank of the node is 0
}
```

```
// Union of two sets

void UNION(edge e)
{
  if (e.source != e.dest)
  {
    LINK(FIND_SET(&vertices[e.source]), FIND_SET(&vertices[e.dest]));
  }
}
```

```
// Find the representative element
node *FIND_SET(node *p)
{
  if (p != p->parent)
    p->parent = FIND_SET(p->parent);
  return p->parent;
}
```

```
struct node
{
  node *parent;   // parent of the node
  long long rank; // rank of the node
};

struct edge
{
  long long source;
  long long dest;
};

node vertices[number_vertices];
```

```
// Link two nodes based on their rank

void LINK(node *x, node *y)
{
  if (x->rank > y->rank)
  {
    y->parent = x;
  }
  else if (x->rank < y->rank)
  {
    x->parent = y;
  }
  else
  {
    y->parent = x;
    x->rank++;
  }
}
```

Class Based Implementation in C++

```cpp
class DSU
{
  vector<int> parent;
  vector<int> rank;

  DSU(int n) // n->number of nodes[0-based indexing]  /** For 1 based indexing size should be n+1
  {
    rank.resize(n, 0);
    parent.resize(n);
    for (int i = 0; i < n; i++)
    {
      parent[i] = i;
    }
  }

  int findParent(int node)
  {
    if (node == findParent(node))
    {
      return node;
    }
    else
    {
      return parent[node] = findParent(parent[node]); // path compression
    }
  }

  void unionByRank(int u, int v)
  {
    int u_p = findParent(u);
    int v_p = findParent(v);
    if (rank[u_p] > rank[v_p])
    {
      parent[v_p] = u_p;
    }
    else if (rank[v_p] > rank[u_p])
    {
      parent[u_p] = v_p;
    }
    else
    {
      parent[u_p] = v_p;
      rank[v_p]++;
    }
  }
};
```

Observation of Union-Find Operations on the dataset

- SNAP Facebook (vertices: 4039, edges: 88234)
  a. Tree Based: 252.87 ms
  b. Linked List Based: 321.124 ms
- SNAP Youtube (vertices: 1134890, edges: 2987624)
  a. Tree Based: 3249.98 ms (3.3 sec)
  b. Linked List Based: 1202489 ms (20.04 min)
- SNAP EU-Core Email (vertices:1005 , edges: 25571)
  a. Tree Based: 50.86 ms
  b. Linked List Based: 198.61ms
- KONECT Full-USA (vertices: 23947347, edges: 57708624)
- a. Tree Based: 81014 ms (1.35 min)
- b. Linked List Based: Ran for approx. 1.5 hours, to reach 0.3% completion

Implementation of Kruskal MST

```c
void kruskal_mst(Graph *graph) {
    int num_vertices = MAX_MEMBERS;
    subset *subsets = (subset *)malloc(num_vertices * sizeof(subset));

    for (int i = 0; i < num_vertices; i++) {
        subsets[i].id = i;
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    Edge *mst = (Edge *)malloc((num_vertices - 1) * sizeof(Edge));
    int mst_index = 0;

    qsort(graph->edges, graph->num_edges, sizeof(Edge), cmpfunc);

    for (int i = 0; i < graph->num_edges; i++) {
        int x = find(subsets, graph->edges[i].src);
        int y = find(subsets, graph->edges[i].dest);
        if (x != y) {
            mst[mst_index] = graph->edges[i];
            mst_index++;
            union_sets(subsets, x, y);
        }
        if (mst_index == num_vertices - 1) {
            break;
        }
    }

    print_mst(mst, mst_index);

    free(mst);
    free(subsets);
}
```

```c
void print_mst(Edge *mst, int num_edges) {
    int total_weight = 0;
    printf("Minimum Spanning Tree:\n");
    for (int i = 0; i < num_edges; i++) {
        printf("( %d , %d )\n", mst[i].src, mst[i].dest);
        total_weight += mst[i].weight;
    }
    //printf("Total weight: %d\n", total_weight);
}
```

Observations

- The previous implementation of connected components and Kruskal MST was performed on the EU-Email dataset.
- The observations for connected components can be found here and for MST can be found here
- From the above implementations we can see that the array based implementation of Disjoint Set Data-Structure provides faster results compared to the linked-list based.


THANK
YOU!
www.funimada.com