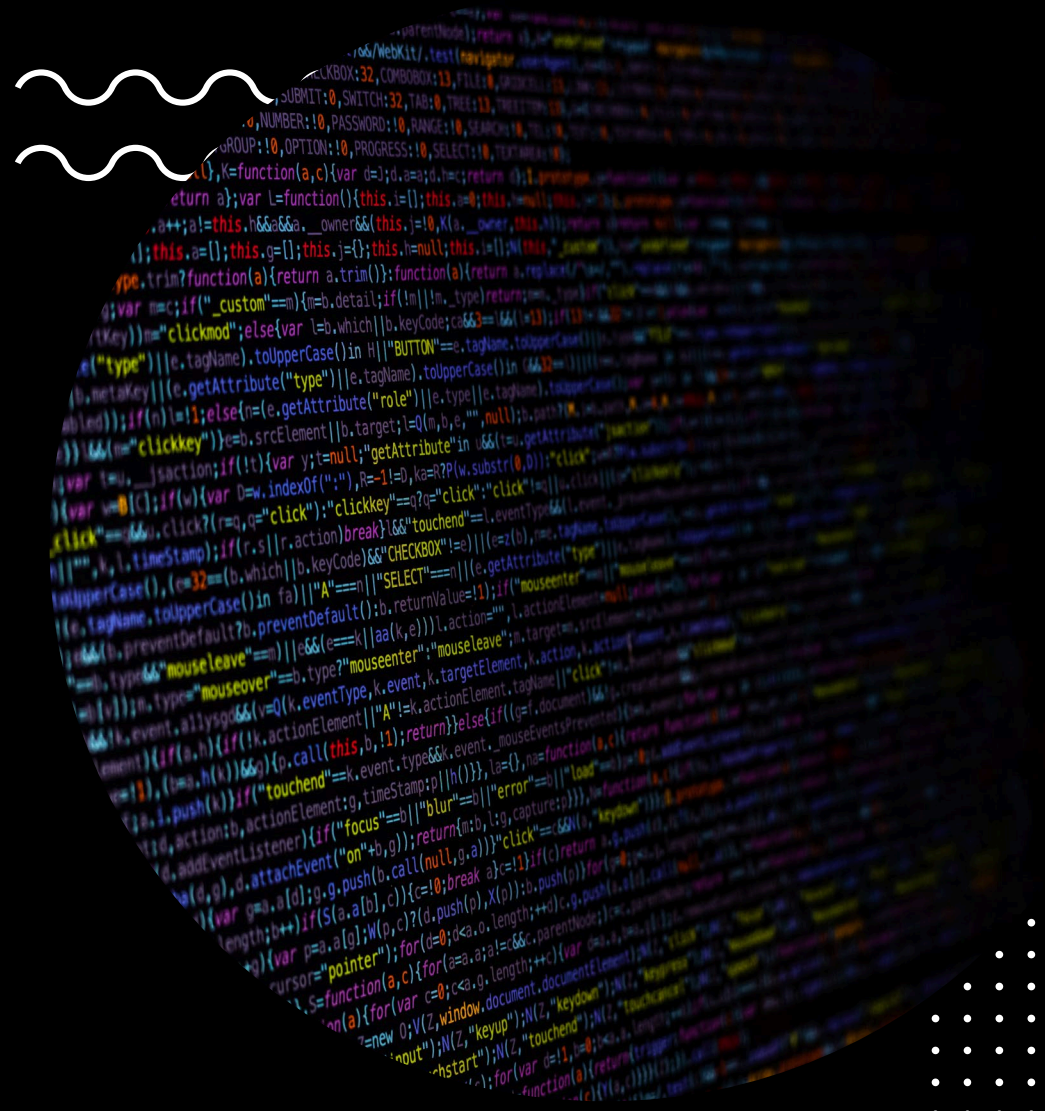
ALGORITHMS LAB (CS-2271) ASSIGNMENT - 2

ARITRA BANDYOPADHYAY
(2021CSB107)

BIPRADEEP BERA
(2021CSB109)



POLYGON TRIANGULATION PROBLEM

1-A: PREPARE A DATASET FOR THE CONVEX POLYGON WITH INCREASING NUMBER OF ARBITRARY VERTICES

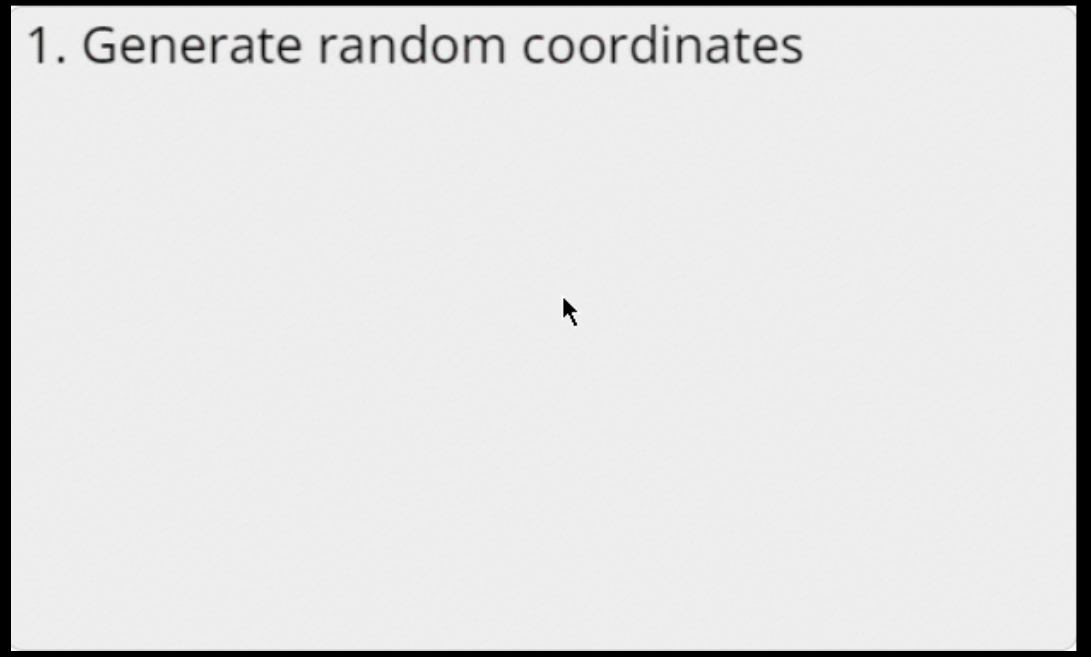
APPROACH FOR GENERATING DATASET

- Here, we will restrict ourselves to convex polygons and we would like each polygon to be equally likely.
- There are many ways to generate convex polygons that result in biased distributions. For example, generating a lot of points at random inside a circle and taking their convex hull gives a random convex polygon. But it will probably look a lot like a circle, whereas most convex polygons do not.
- Reference: https://cglab.ca/~sander/misc/ConvexGeneration/convex.html

APPROACH FOR GENERATING DATASET

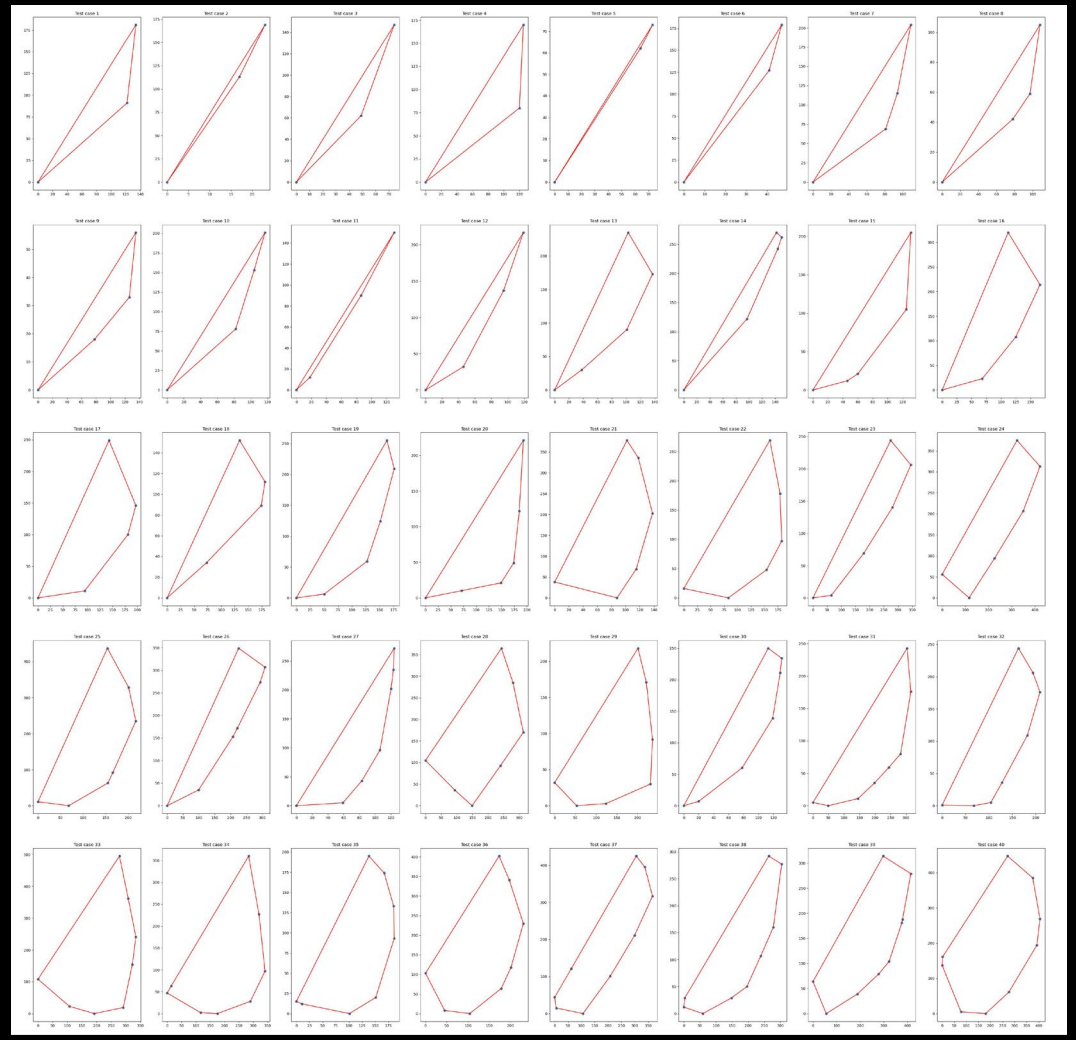
ALGORITHM:

1. Generate two lists of random X and Y coordinates
2. Sort them
3. Isolate the extreme points
4. Randomly divide the interior points into two chains
5. Extract the vector components
6. Randomly pair up the X- and Y-components
7. Combine the paired up components into vectors
8. Sort the vectors by angle
9. Lay them end-to-end to form a polygon
10. Move the polygon to the original min and max coordinates



PLOTTING A SUBSET OF DATASET

We generated 30 polygons with number of vertices ranging from 3 (triangles) to 90.



1-B: CONSIDER A BRUTE FORCE METHOD WHERE YOU DO AN EXHAUSTIVE SEARCH FOR FINDING THE OPTIMAL RESULT

BRUTE FORCE APPROACH

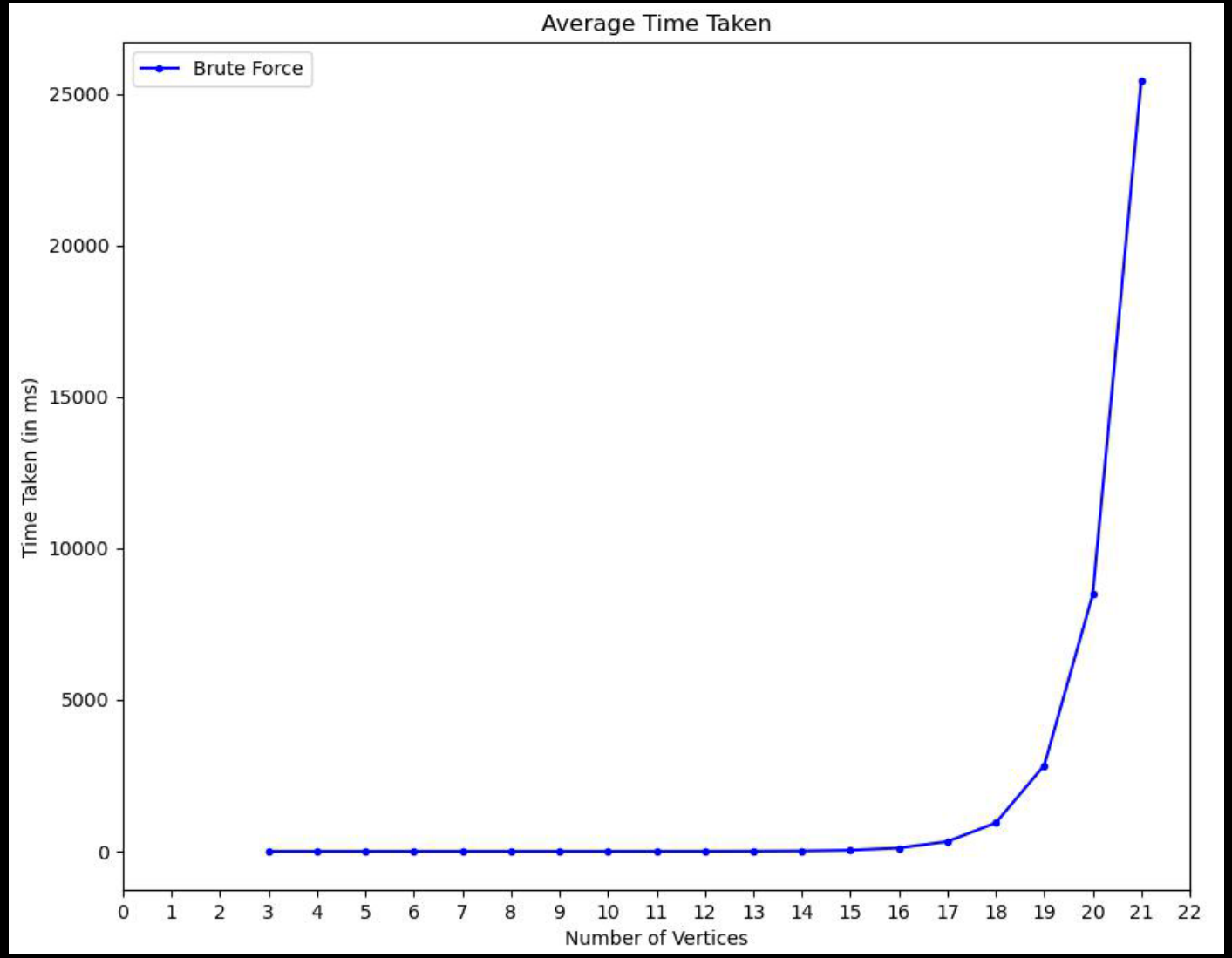
- A triangulation of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners) such that the diagonals never intersect. The problem is to find the cost of triangulation with the minimum cost. The cost of a triangulation is sum of the weights of its sides (perimeter).
- A polygon with n vertices can be triangulated into n-2 triangles and we have to find min cost for such triangulation.
- This problem has **recursive substructure**. The idea is to divide the polygon into three parts: a single triangle, the sub-polygon to the left, and the sub-polygon to the right. We try all possible divisions like this and find the one that minimizes the cost of the triangle plus the cost of the triangulation of the two sub-polygons.

CODE FOR BRUTE FORCE APPROACH

Here cost is a helper function that defines the cost for a triangulation to be the perimeter of the triangle formed by the 3 points

```
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// Recursively find min triangulation cost
// The polygon is represented by points[i..j].
double mTC(Point points[], int i, int j)
{
    // There must be at least three points between i and j
    // (including i and j)
    if (j < i + 2)
        return 0;

    // Initialize result as infinite
    double res = MAX;

    // Find minimum triangulation by considering all
    for (int k = i + 1; k < j; k++)
        res = min(res, (mTC(points, i, k) + mTC(points, k, j) + cost(points, i, k, j)));
    return res;
}
```

ANALYSIS OF BRUTE FORCE APPROACH

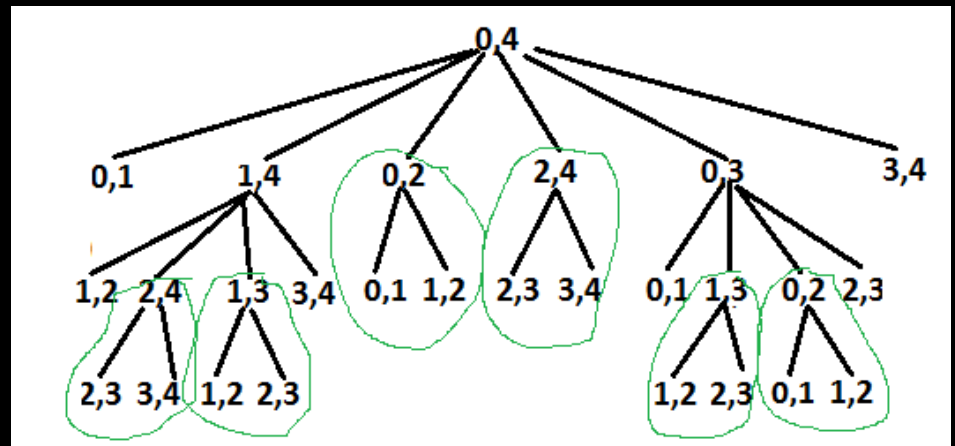


ANALYSIS OF BRUTE FORCE APPROACH

- The time taken by the naïve recursive approach varies exponentially as the number of vertices increases.
- Theoretically, this is because the problem size varies with Catalan Numbers as n (number of vertices) increases:

$$\frac{4^n}{n^{3/2}}$$

- This is because we are solving subproblems again and again when we encounter overlapping problems. Thus we can optimize by keeping a track of these solutions and not solving them again.

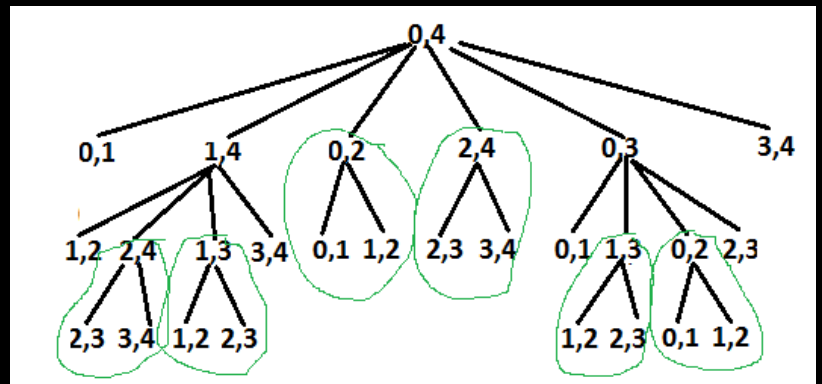


Recursion Tree for recursive implementation. Overlapping subproblems are encircled.

1-C: NEXT APPLY DYNAMIC PROGRAMMING, IN LINE WITH THE MATRIX CHAIN MULTIPLICATION PROBLEM

DYNAMIC PROGRAMMING APPROACH

- It can be easily seen in the recursion tree that the problem has many overlapping subproblems. Since the problem has both properties: **Optimal Substructure** and **Overlapping Subproblems**, it can be efficiently solved using Dynamic Programming.
- We will be storing the solution for the subproblems in a dp table and when overlapping problems come up while solving larger problems, we can simply use the result from the table.



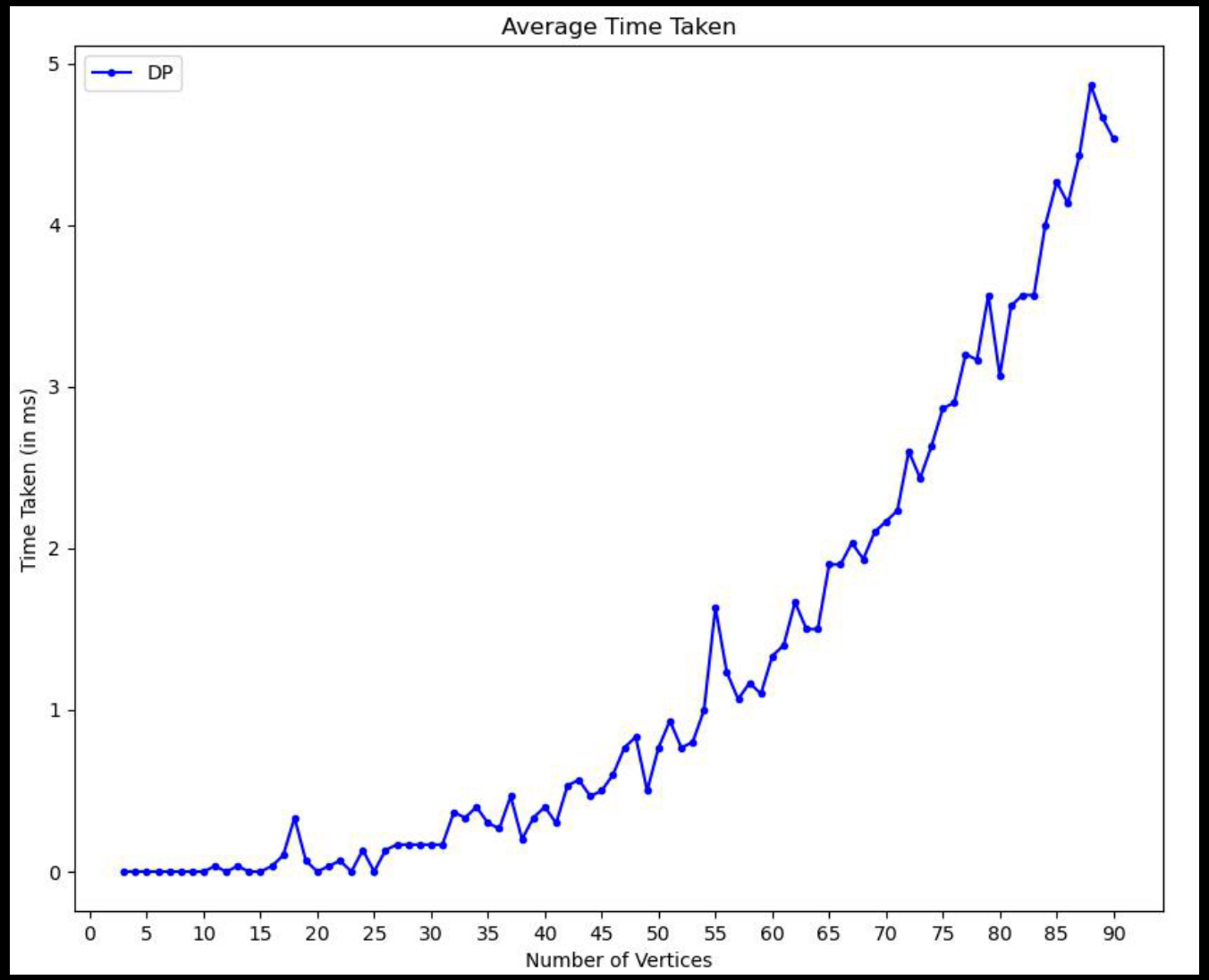
Recursion Tree for recursive implementation. Overlapping subproblems are encircled.

CODE FOR DP APPROACH

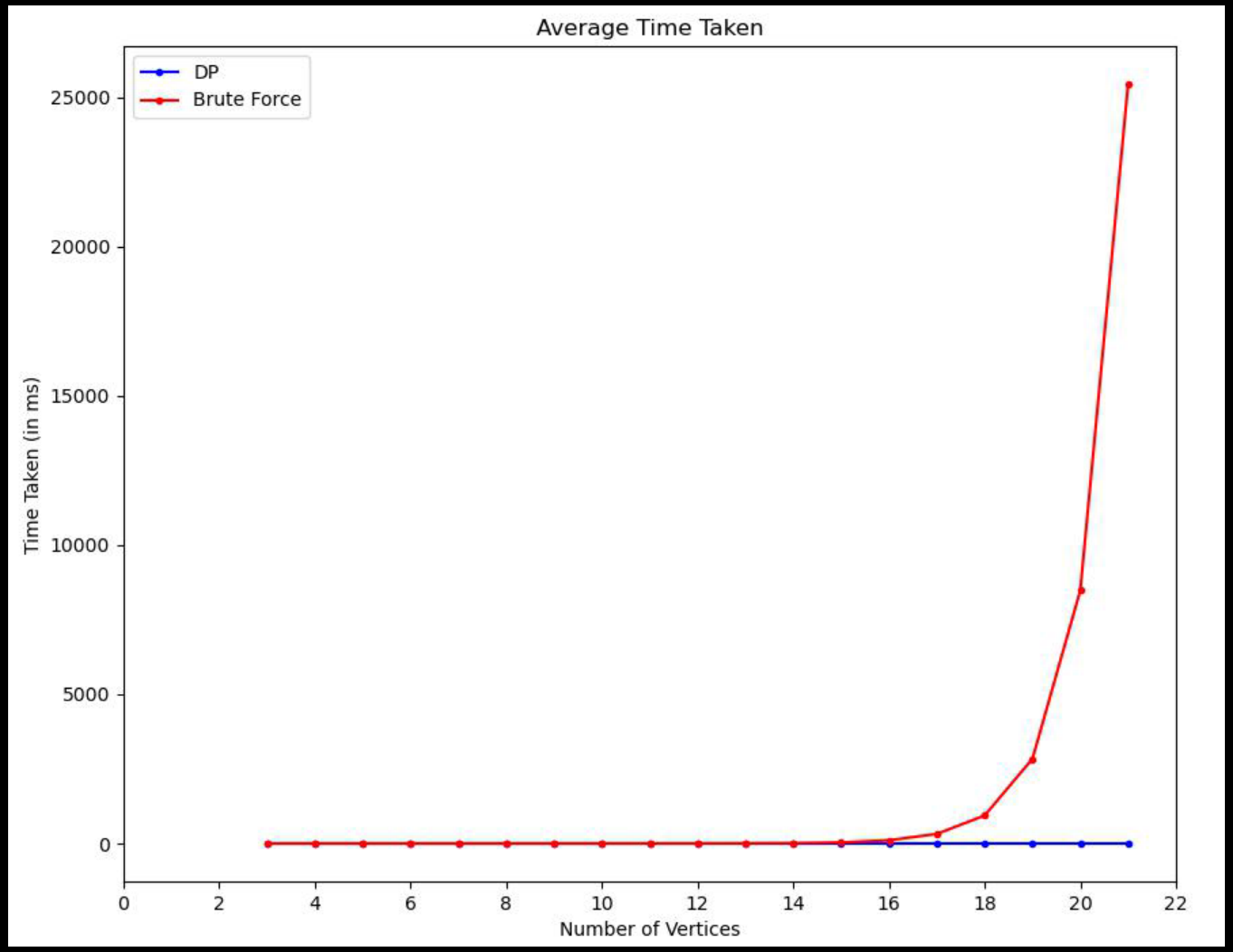
- Here the cost function is the same as that defined previously.
- dp[i][j] stores the cost for triangulating points i to j.
- dp[0][n – 1] is the final result for triangulating n points.

```
double mTCDP(Point points[], int n) {
    // Base case, if less than 3 points
    if (n < 3) return 0;
    // dp[i][j] stores cost of triangulation of points from i to j
    // dp[0][n-1] is the final answer, considering n points
    double dp[n][n];

    // dp is filled in diagonal fashion
    for (int gap = 0; gap < n; gap++) {
        for (int i = 0, j = gap; j < n; i++, j++) {
            if (j < i+2) dp[i][j] = 0.0;
            else {
                dp[i][j] = MAX;
                for (int k = i+1; k < j; k++) {
                    double val = dp[i][k] + dp[k][j] + cost(points,i,j,k);
                    if (dp[i][j] > val) dp[i][j] = val;
                }
            }
        }
    }
    return dp[0][n-1];
}
```

ANALYSIS OF DP APPROACH



DP VS BRUTE FORCE APPROACH



ANALYSIS OF DP APPROACH

- The time complexity of the dynamic programming approach is $O(n^3)$.
- Auxiliary Space required is $O(n^2)$ as we are storing the results of the subproblems in a 2D array.
- This is significantly better than the exponential Brute Force approach.

1-D: THEN IMPLEMENT A GREEDY STRATEGY TO SOLVE THE SAME PROBLEM BY CHOOSING NON-INTERSECTING DIAGONALS IN SORTED ORDER

GREEDY APPROACH

- This approach works by clipping ears from the polygon and drawing a diagonal in its place and slowly reducing the polygon, until full triangulation is achieved.
- This algorithm runs in $O(n^2)$ time and is thus faster than the DP approach.
- However, this might not give optimal triangulation as it does not guarantee that adding triangles with smaller perimeters will give an overall optimal solution

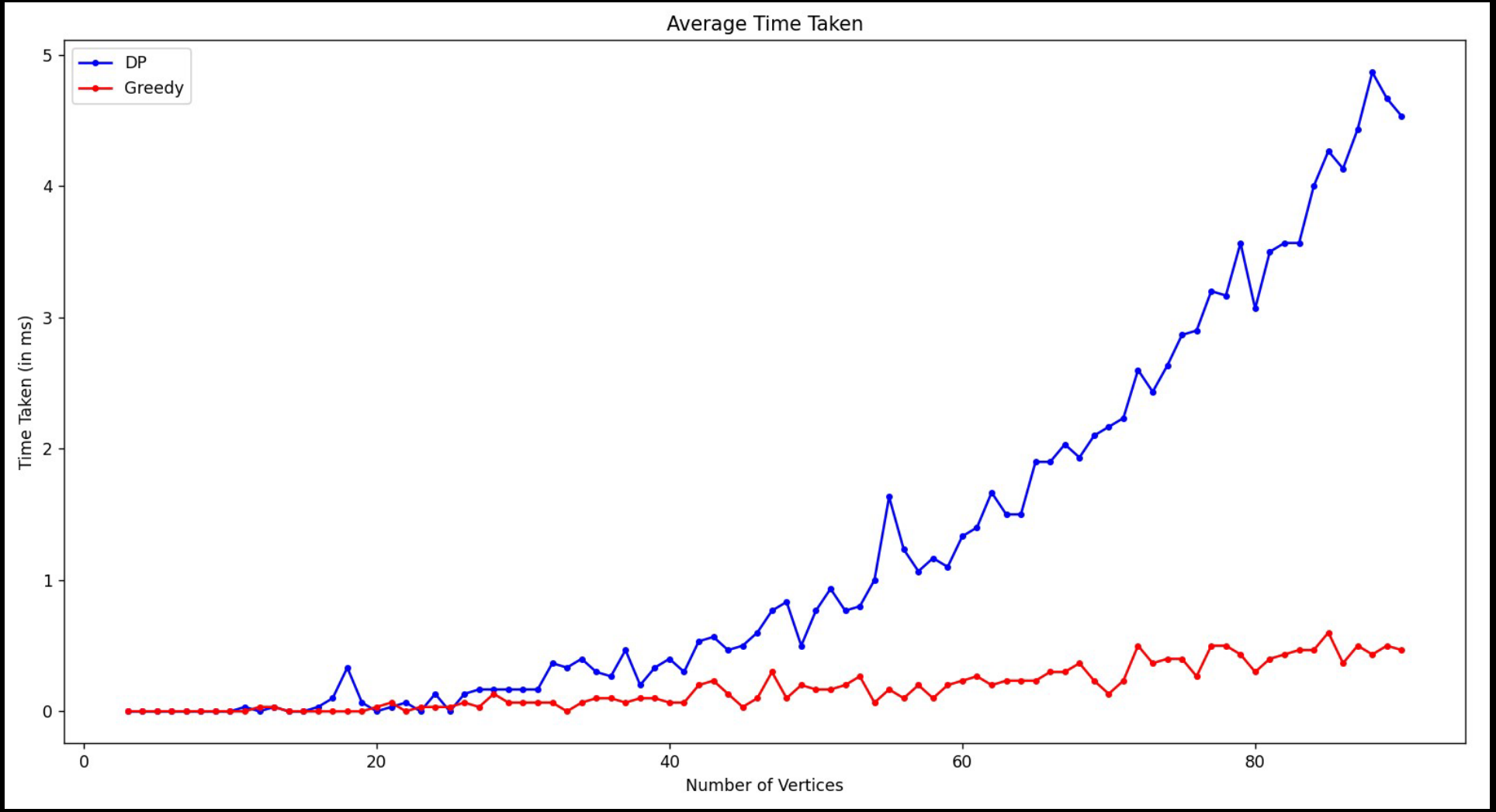
CODE FOR GREEDY APPROACH

- Each call does at most O(n) work and there will be n-2 calls to fully triangulate the polygon, which results in $O(n^2)$ complexity.
- This may not result in optimal triangulation because iteratively adding smaller triangles to our solution doesn't guarantee overall small cost.

```
double mTCG(Point *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 + mTCG(points, newVisited, n - 1);
}
```

1-E: FINALLY COMPARE THE RESULTS AND SUGGEST WHETHER THE DP AND GREEDY APPROACH RESULTS SHOW MISMATCH

DP VS GREEDY APPROACH

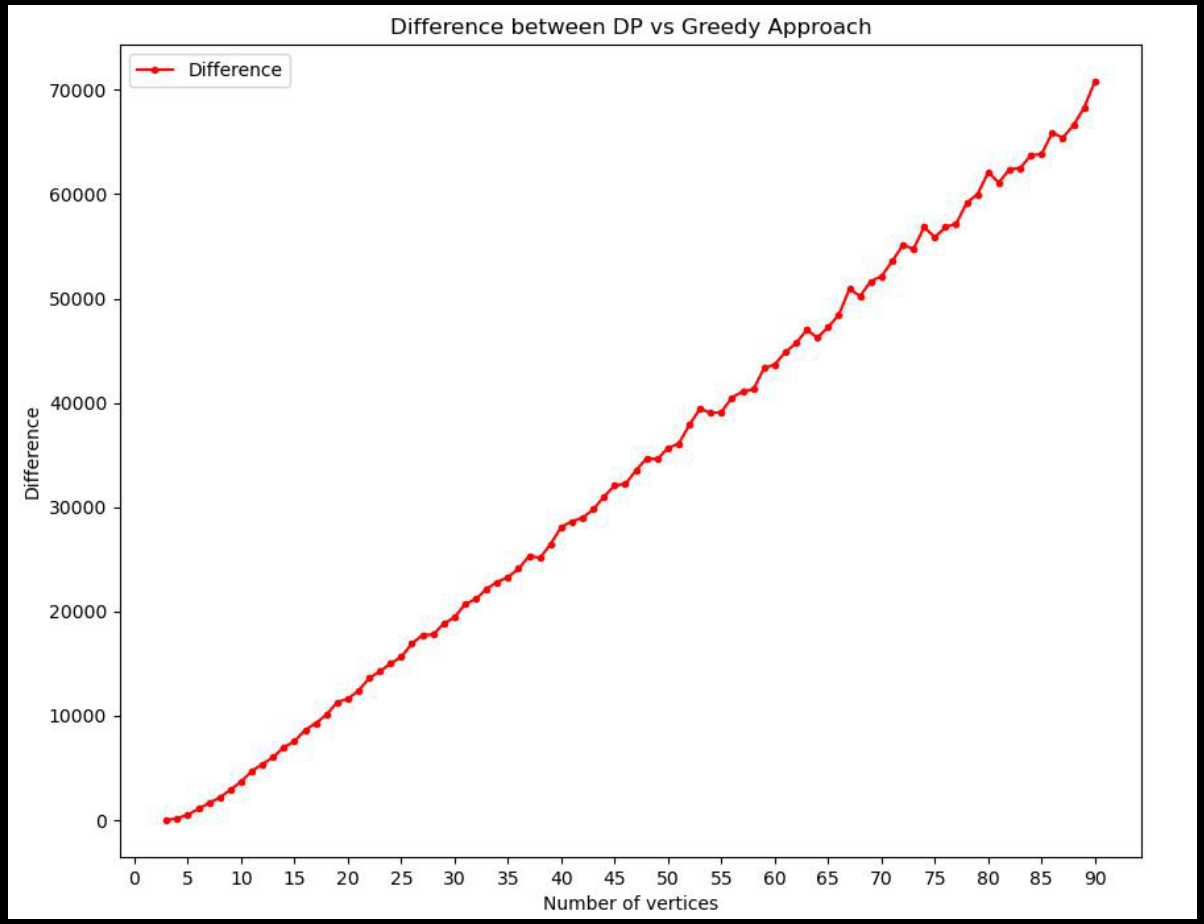


ANALYSIS OF GREEDY VS DP APPROACH

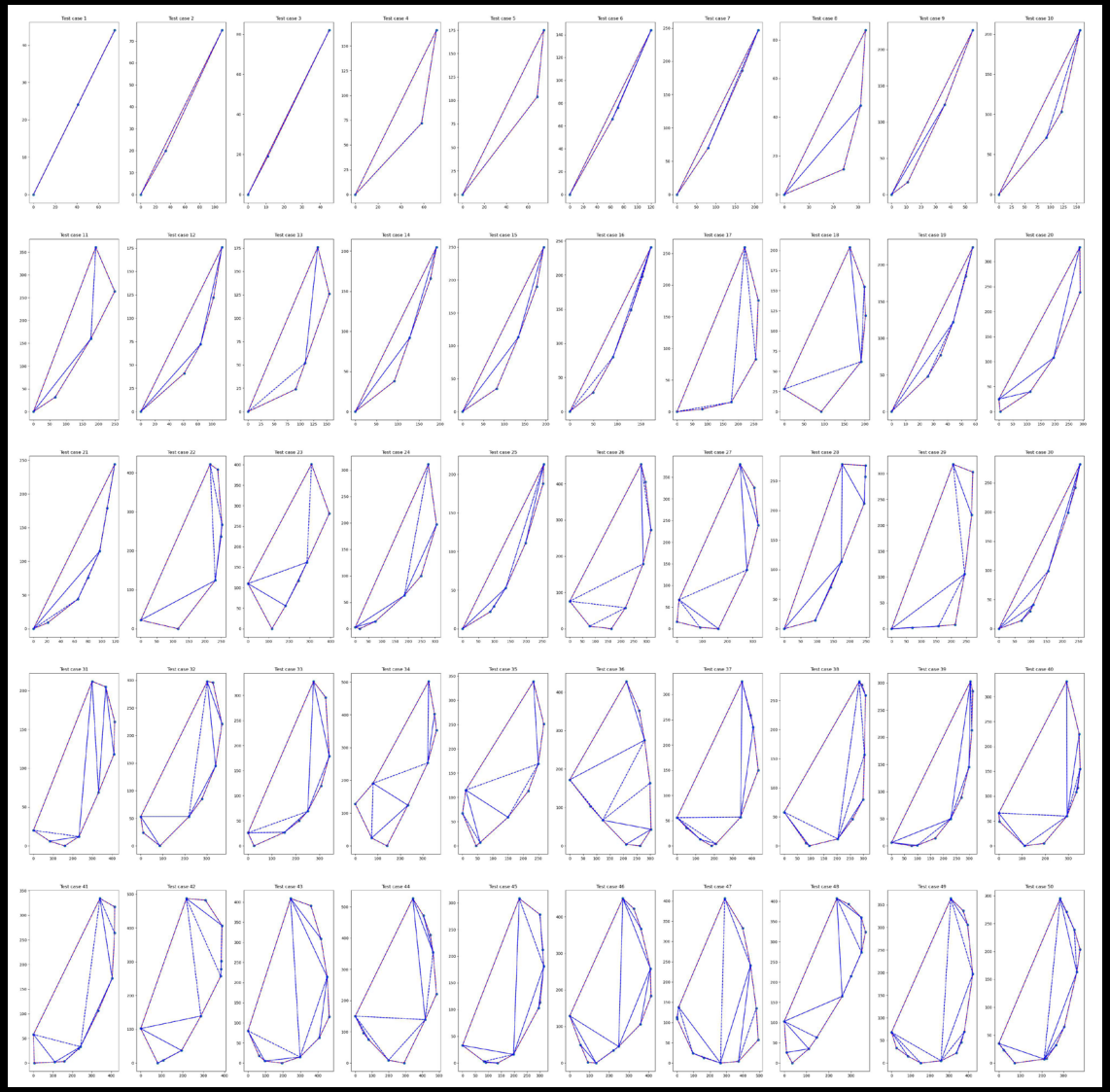
- From the graph, it is clear that the Greedy Approach grows more slowly than the DP one as the number of vertices increases.
- This is due to the fact that the Greedy Strategy works in $O(n^2)$ time while the DP one takes $O(n^3)$ time

DIFFERENCE IN GREEDY VS DP APPROACH

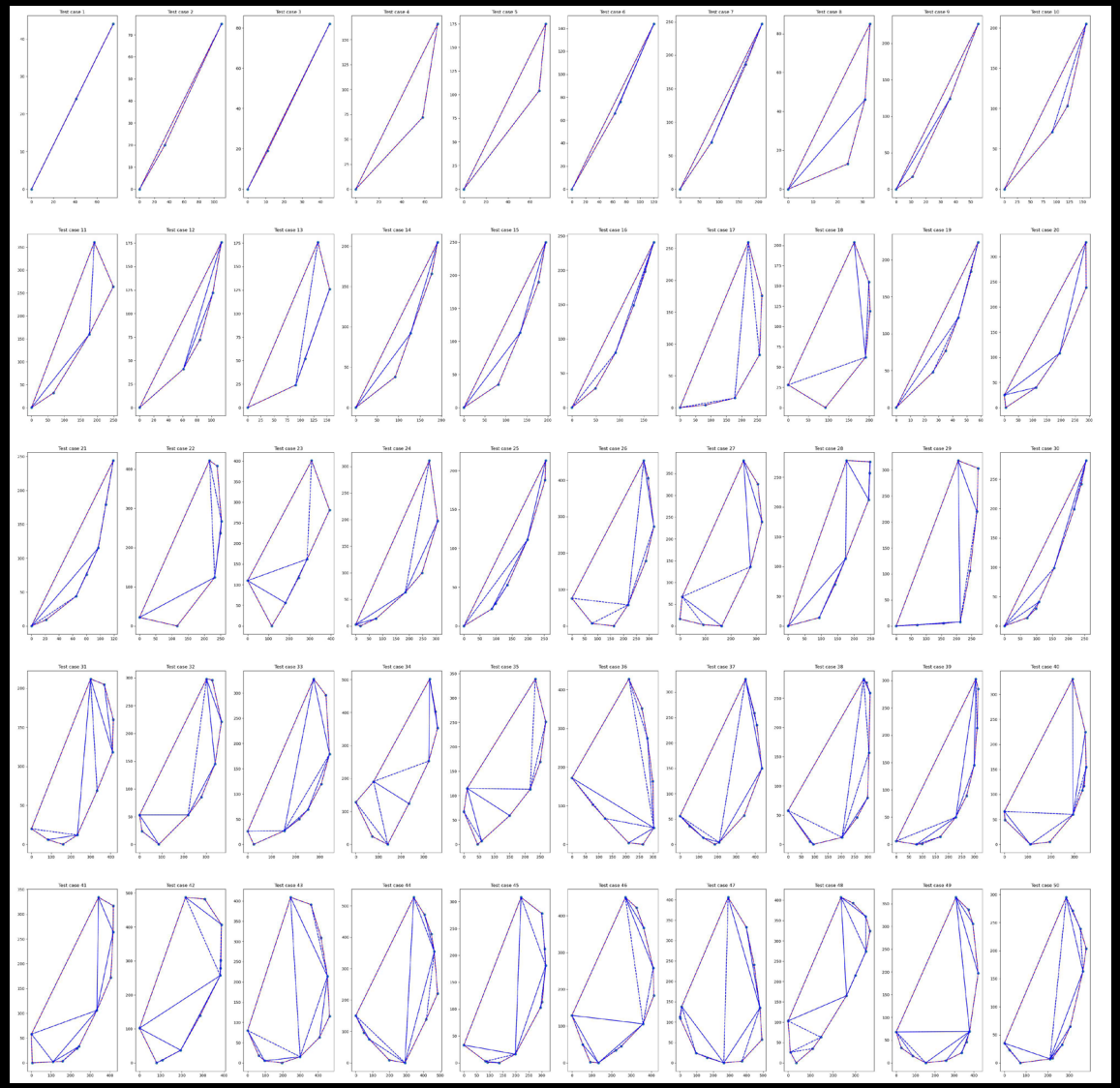
- As we can see, the average difference between the DP vs Greedy Approaches (averaged over the 30 polygons for each n) increases as n increases
- Error due to this approach vs DP optimal triangulation will also depend on how the polygons were generated.



DP TRIANGULATION PLOTS



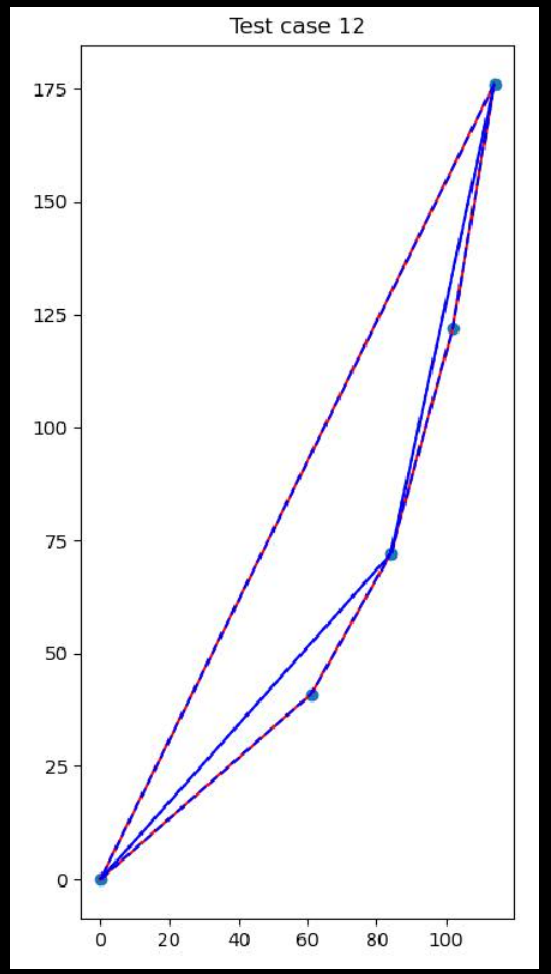
GREEDY TRIANGULATION PLOTS



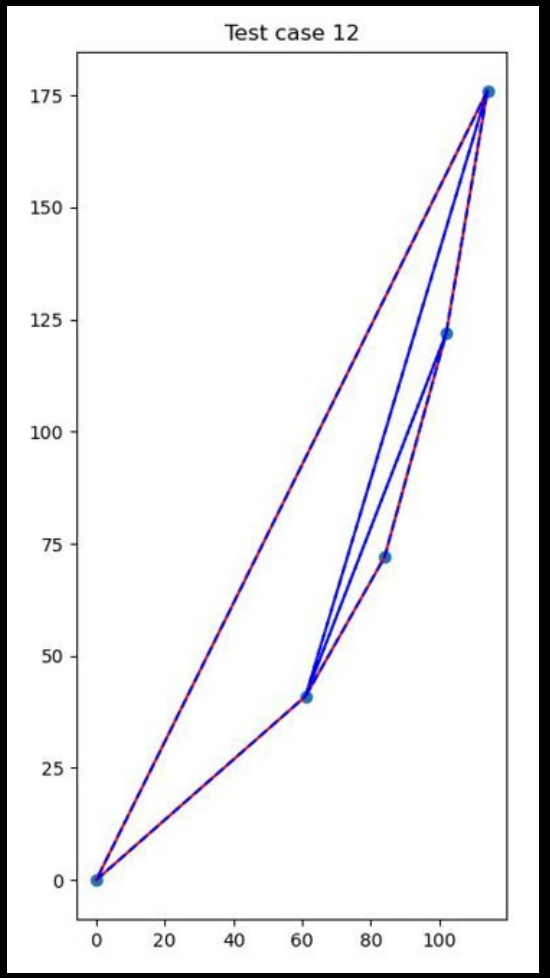
COMPARISON OF TRIANGULATIONS

Comparison of triangulation for a testcase with number of vertices = 5

The greedy method doesn't always provide wrong answer (specially for less vertices) but as n increases, average error over many polygons increases and tends to vary linearly in our case (will depend on data generation method too)

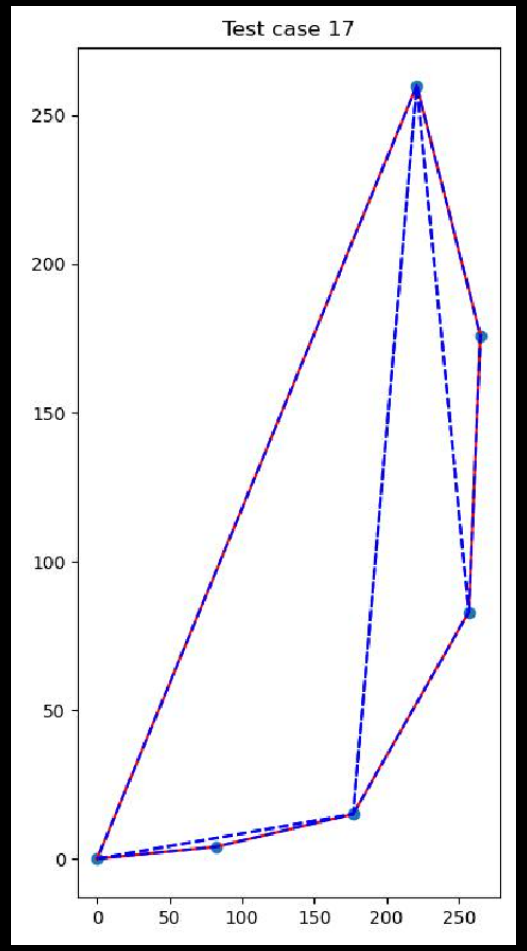


DP Triangulation (868.002)

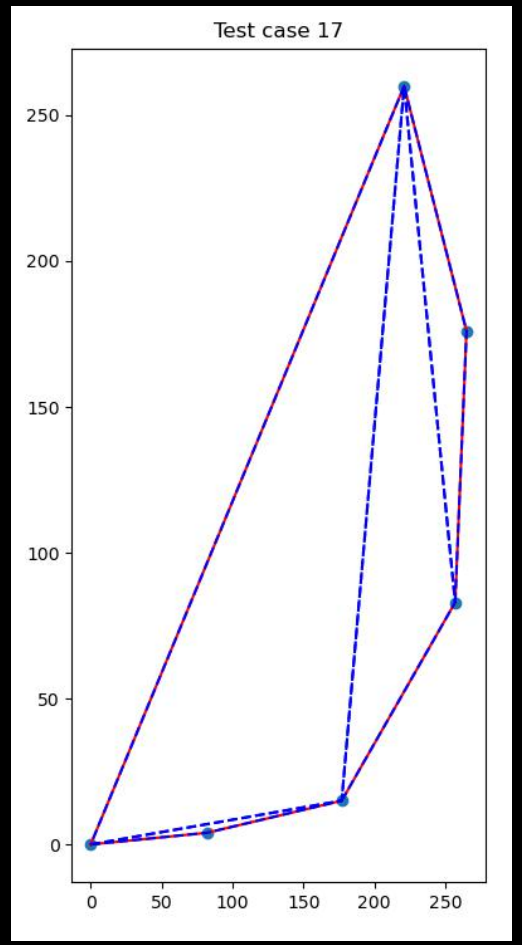


Greedy Triangulation (1284.67)

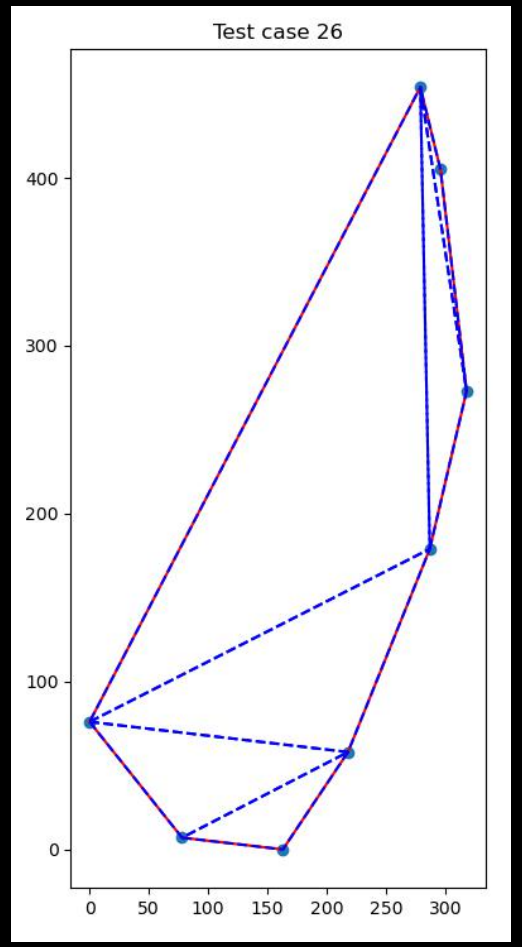
COMPARISON OF TRIANGULATIONS



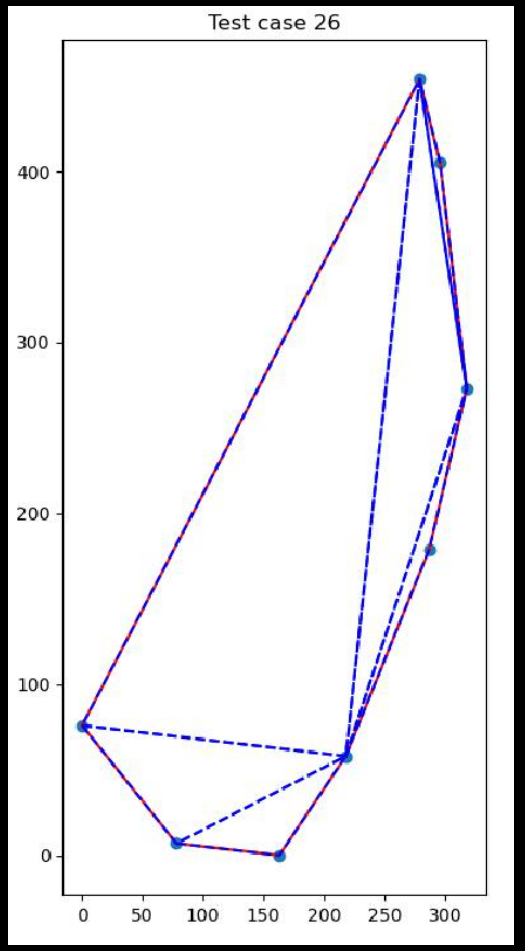
DP Triangulation



Greedy Triangulation



DP Triangulation



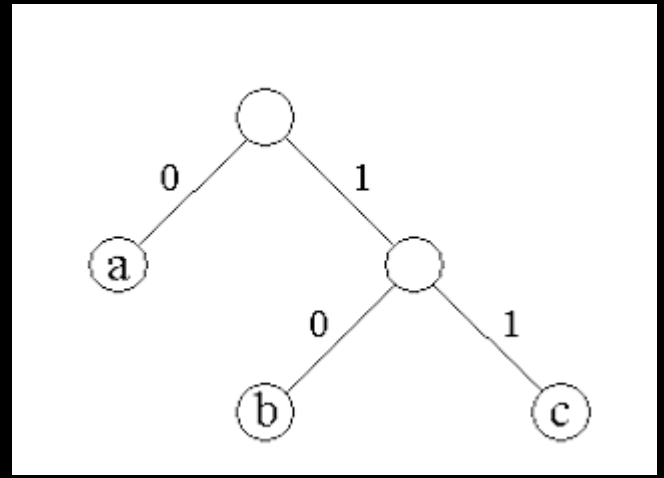
Greedy Triangulation

DATA COMPRESSION STRATEGIES

2-A: WRITE ENCODING ALGORITHMS FOR THE SHANNON-FANO CODING SCHEME

DATA COMPRESSION

- Data Compression, also known as source coding, is the process of encoding or converting data in such a way that it consumes less memory space. Data compression reduces the number of resources required to store and transmit data.
- A prefix code is a type of code system distinguished by its possession of the "prefix property", which requires that there is no whole code word in the system that is a prefix (initial segment) of any other code word in the system.
- Prefix codes are also known as instantaneous codes.
- Code trees consist of interior nodes, leaf nodes and corresponding branches. Leaf nodes do not have a succeeding node and represent symbols, if the tree describes a prefix code. The path from the root to a leaf node defines the code word for the particular symbol assigned to this node.



Coding Trees

SHANNON-FANO CODING SCHEME

- Shannon Fano Algorithm is an entropy encoding technique for lossless data compression of multimedia. It assigns a code to each symbol based on their probabilities of occurrence.
- It is a variable-length encoding scheme, that is, the codes assigned to the symbols will be of varying lengths.
- It is a prefix-coding scheme.
- Shannon–Fano codes are suboptimal in the sense that they do not always achieve the lowest possible expected codeword length, as Huffman coding does, but the Shannon-Fano codewords have an expected codeword length within 1 bit of the optimal.

a_i	$p(a_i)$	1	2	3	4	Code
a_1	0.36	0	00			00
a_2	0.18		01			01
a_3	0.18	1	10			10
a_4	0.12		11	110		110
a_5	0.09			111	1110	1110
a_6	0.07				1111	1111

CODE FOR SHANNON-FANO CODING

- This is a recursive implementation of the Shannon-Fano algorithm for assigning binary codes to a sorted list of symbols based on their probabilities.
- The function sums probabilities for symbols in [startIndex, endIndex], then finds ‘partitionIndex’ that splits this range into two sub-ranges. It ensures the first sub-range has probabilities closest to half of the total sum.
- The function then assigns the binary code '0' to all symbols in the first sub-range and the binary code '1' to all symbols in the second sub-range.
- The function then recursively calls itself twice, passing the first and second sub-ranges as the new ranges to be processed.

```
void shannonFano(vector<Symbol>& symbols, int startIndex, int endIndex) {
    if (startIndex == endIndex) {
        return;
    }

    double sumProb = 0.0;
    for (int i = startIndex; i <= endIndex; i++) {
        sumProb += symbols[i].probability;
    }

    double halfProb = 0.0;
    int partitionIndex = -1;
    for (int i = startIndex; i <= endIndex; i++) {
        halfProb += symbols[i].probability;
        if (halfProb >= sumProb / 2) {
            partitionIndex = i;
            break;
        }
    }

    for (int i = startIndex; i <= partitionIndex; i++) {
        symbols[i].code += '0';
    }

    for (int i = partitionIndex + 1; i <= endIndex; i++) {
        symbols[i].code += '1';
    }

    shannonFano(symbols, startIndex, partitionIndex);
    shannonFano(symbols, partitionIndex + 1, endIndex);
}
```

SHANNON-FANO ENCODING SCHEME

- The Shannon-Fano code word for the 6 symbols, A, B, C, D, E, F are shown with the probabilities as follows:

  A: 0.3

  B: 0.25

  C: 0.15

  D: 0.12

  E: 0.08

  F: 0.10

```
Enter the number of symbols: 6
Enter symbol 1: A
Enter probability of A: 0.3
Enter symbol 2: B
Enter probability of B: 0.25
Enter symbol 3: C
Enter probability of C: 0.15
Enter symbol 4: D
Enter probability of D: 0.12
Enter symbol 5: F
Enter probability of F: 0.10
Enter symbol 6: E
Enter probability of E: 0.08

Symbol  Probability     Code
A       0.3             00
B       0.25            01
C       0.15            100
D       0.12            101
F       0.1             110
E       0.08            111
```

2-B: IMPLEMENT HUFFMAN ENCODING SCHEME USING HEAP AS DATA STRUCTURE

HUFFMAN ENCODING SCHEME

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
- It is a greedy algorithm that constructs an optimal prefix code.
- C is a set of n characters and each character c ∈ C is an object with an attribute c.freq giving its frequency.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| - 1 "merging" operations to create the final tree.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

CODE FOR HUFFMAN ENCODING

- The **buildHuffmanTree** function takes a string as input and creates a map to store the frequency of each symbol in the string. It then creates a priority queue (min-heap) to store the Huffman nodes. The Huffman nodes are created for each symbol with its frequency and added to the priority queue.
- Then, the function builds the Huffman tree by extracting the two minimum frequency nodes from the priority queue and creating a new node with the sum of the frequencies of the two nodes.
- This new node is added back to the priority queue, and the process is repeated until there is only one node left in the priority queue, which is the root node of the Huffman tree.

```
// Function to build the Huffman tree
HuffmanNode* buildHuffmanTree(string text) {
    // Create a map to store the frequency of each symbol
    unordered_map<char, int> freq;
    for (char c : text) {
        freq[c]++;
    }

    // Create a priority queue (min-heap) to store the Huffman nodes
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, CompareHuffmanNode> pq;
    for (auto& f : freq) {
        pq.push(new HuffmanNode(f.first, f.second));
    }

    // Build the Huffman tree
    while (pq.size() > 1) {
        // Extract the two minimum frequency nodes from the priority queue
        HuffmanNode* left = pq.top();
        pq.pop();
        HuffmanNode* right = pq.top();
        pq.pop();

        // Create a new node with the sum of the frequencies of the two nodes
        HuffmanNode* newNode = new HuffmanNode('\0', left->frequency + right->frequency);
        newNode->left = left;
        newNode->right = right;

        // Add the new node to the priority queue
        pq.push(newNode);
    }

    // Return the root node of the Huffman tree
    return pq.top();
}
```

CODE FOR HUFFMAN ENCODING (contd.)

- **encodeText** function takes the input text and retrieves its corresponding map from the code and the final string is returned as encodedText.
- **assignHuffmanCode** The function first checks if the current node is a leaf node. If yes, the function stores the Huffman code generated so far for the symbol represented by the node in the unordered map. Otherwise, the function recursively calls itself for the left and right child of the current node. When calling itself for the left child, it appends "0" to the current code, and when calling itself for the right child, it appends "1" to the current code. This process continues until all the leaf nodes in the Huffman tree have been processed and their corresponding Huffman codes have been stored in the unordered map.

```
// Function to encode the text using the Huffman tree
string encodeText(string text, unordered_map<char, string>& code) {
    string encodedText = "";
    for (char c : text) {
        encodedText += code[c];
    }
    return encodedText;
}

// Function to traverse the Huffman tree and assign codes to symbols
void assignHuffmanCodes(HuffmanNode* node, string code, unordered_map<char, string>& huffmanCode) {
    if (node->left == nullptr && node->right == nullptr) {
        huffmanCode[node->symbol] = code;
    }
    else {
        assignHuffmanCodes(node->left, code + "0", huffmanCode);
        assignHuffmanCodes(node->right, code + "1", huffmanCode);
    }
}
```

EXAMPLE OUTPUT OF HUFFMAN CODE

This shows the Huffman encoding for the string “Hello, world!”

NOTE: Huffman codes are not unique, so there may be many representations that are all optimal.

```
Huffman Codes:
 : 1111
d: 1110
w: 1010
,: 1001
o: 110
H: 1011
!: 1000
l: 01
r: 001
e: 000
Input Text: Hello, world!
Encoded Text: 101100001011101001111110101100010111101000
```

2-C: IMPLEMENT AN INSTANTANEOUS DECODING ALGORITHM

INSTANTANEOUS DECODING

- A code is a prefix code if no target bit string in the mapping is a prefix of the target bit string of a different source symbol in the same mapping.
- This means that symbols can be decoded instantaneously after their entire codeword is received.
- Huffman codes are examples of such code and they can be instantaneously decoded.

CODE FOR INSTANTANEOUS DECODING

- To decode the encoded text, we need to use the same Huffman tree that was used for encoding. Each symbol in the encoded text corresponds to a unique path from the root of the Huffman tree to a leaf node. We start at the root node of the Huffman tree and traverse it according to the encoded text.
- For each bit in the encoded text, if it is a 0, we move to the left child of the current node in the Huffman tree, else, we move to the right child. We continue this process until we reach a leaf node, which corresponds to a decoded symbol. We add this symbol to the decoded text and reset the current node to the root of the Huffman tree.
- We repeat this process until all bits in the encoded text have been decoded into symbols.

```
// Function to decode the text using the Huffman tree
string decodeText(string encodedText, HuffmanNode* root) {
    string decodedText = "";
    HuffmanNode* current = root;
    for (char c : encodedText) {
        if (c == '0') {
            current = current->left;
        }
        else if (c == '1') {
            current = current->right;
        }
        if (current->left == nullptr && current->right == nullptr) {
            decodedText += current->symbol;
            current = root;
        }
    }
    return decodedText;
}
```

EXAMPLE OUTPUT OF DECODING

This shows the Huffman encoding and decoding for the string “Hello, world!”

The text that has been decoded from the codeword exactly matches the input text and thus our implementation is correct.

```
Input Text: Hello, world!
Encoded Text: 1011000010111010011111101011000101111101000
Decoded Text: Hello, world!
```

2-D: CHOOSE A LARGE TEXT DOCUMENT AND GET RESULTS OF HUFFMAN CODING WITH THIS DOCUMENT

HUFFMAN CODING WITH LARGE TEXT DOC

- A large text document (containing 8,685 characters, including spaces) was considered.
- A compression of 43.5% was achieved for this text file using Huffman Coding

```
Compression percentage: 43.85%
```

2-D: EXAMINE THE OPTIMALITY OF HUFFMAN CODES AS DATA COMPRESSION STRATEGY BY COMPARING WITH ANOTHER DOCUMENT

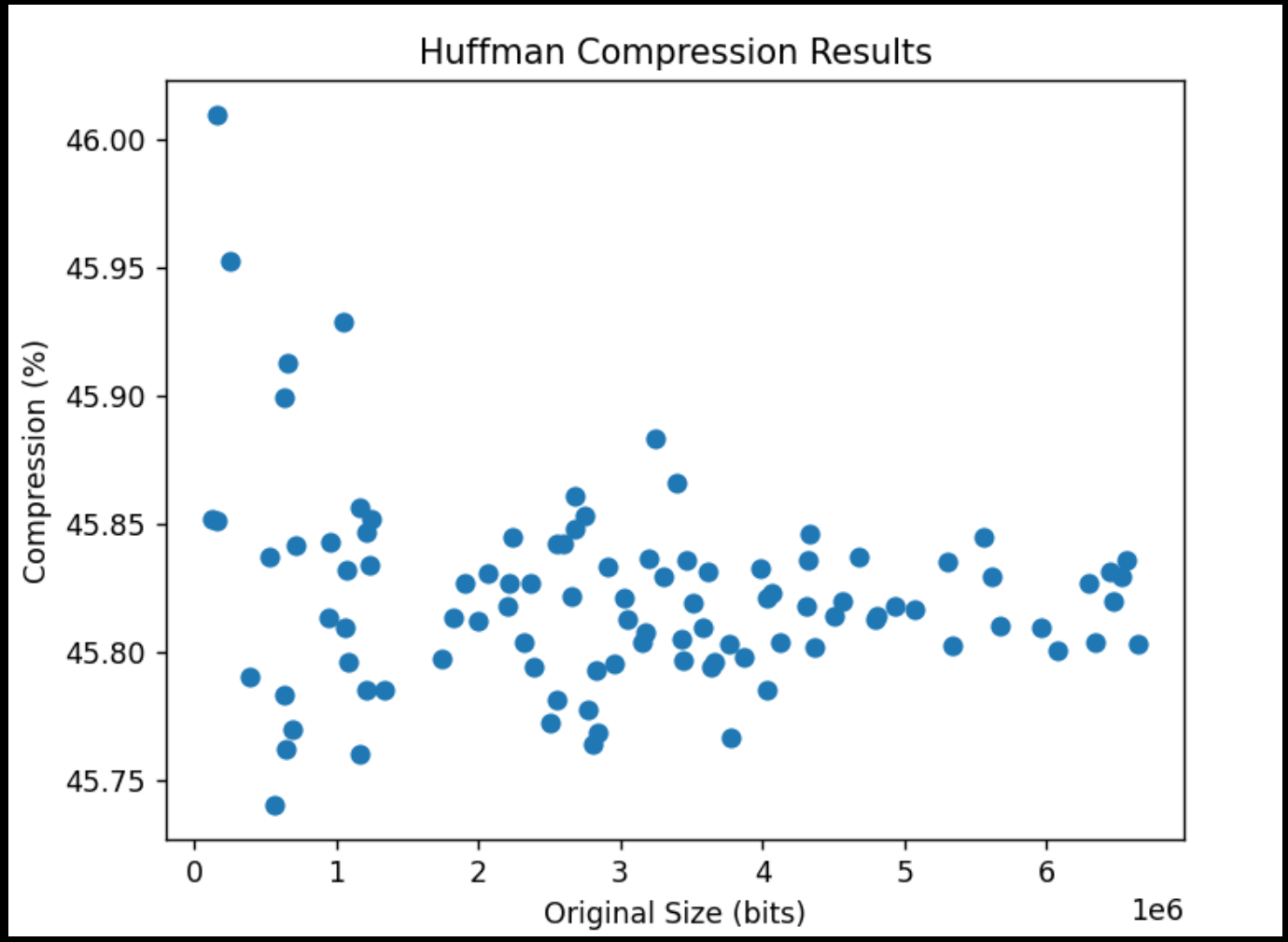
GENERATING DATASET OF LARGE TXT

- 100 large text files were generated using an online API by web scraping using Python

```python
import os
import random
import requests

# Set the number of files to generate and the word count range for each file
num_files = 100
min_ = 20
max_ = 2000

# Create a directory to store the generated text files
if not os.path.exists('generated_text'):
    os.makedirs('generated_text')

# Loop through the number of files to generate
for i in range(num_files):
    # Generate a random file name
    file_name = f'input/input{i}.txt'

    # Generate a random word count for the file
    word_count = random.randint(min_, max_)

    # Generate the text using the "Lorem Ipsum" text generator API
    response = requests.get(f'https://loripsum.net/api/{word_count}/plaintext')

    # Write the generated text to the file
    with open(file_name, 'w') as f:
        f.write(response.text)

    print(f'Generated file {i+1}/{num_files}: {file_name}')
```

PLOT OF HUFFMAN COMPRESSION %

- Using our generated dataset, Huffman coding achieved a compression percentage between 45.5-46.5 for all of the files.



3: STUDY AND IMPLEMENT DSU OPERATIONS FOR SNAP AND KONECT DATASETS

MST AND CONNECTED COMPONENTS

- A Minimum Spanning Tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.
- A Connected Component of an undirected graph is a connected subgraph that is not part of any larger connected subgraph. The components of any graph partition its vertices into disjoint sets, and are the induced subgraphs of those sets. A graph that is itself connected has exactly one component, consisting of the whole graph.

ABOUT THE DATASETS

- 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

LINKED LIST IMPLEMENTATION

- Here we used Two Linked Lists, one for dynamically managing number of Connected Components of the graph (also called Representative 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, which stores the length of vertex linked list inside it.

IMPLEMENTATION OF OPERATIONS

- The disjoint-set data structure has three main operations: Make_Set, Find_Set, and Union.
- Make_Set creates a new representative node with a given vertex and adds it to the existing doubly linked list.
- Find_Set searches through all the representative nodes, checking their values, and returns the representative element if found.
- Union takes two representative nodes as input, merges their vertex lists, updates the representative element of the merged list, and deletes the representative node of the smaller list. The time complexity of these operations is O(log n) on average, where n is the number of unions performed.

CODE FOR IMPLEMENTATION

```
void MAKE_SET(long long x, conn_comp *start) {
    conn_comp *temp = start->next;

    conn_comp *new_conn_comp = new conn_comp;
    new_conn_comp->len = 1;
    new_conn_comp->next = temp;
    start->next = new_conn_comp;
    new_conn_comp->prev = start;

    if (temp != NULL)
        temp->prev = new_conn_comp;

    new_conn_comp->head = new node;
    new_conn_comp->tail = new_conn_comp->head;
    new_conn_comp->head->data = x;
    new_conn_comp->head->rep = new_conn_comp;
    new_conn_comp->head->next = NULL;
}
```

```
conn_comp *FIND_SET(long long val, conn_comp *start) {
    conn_comp *conn_comp_iterator = start->next;
    while (conn_comp_iterator != NULL) {
        node *node_iterator = conn_comp_iterator->head;
        while (node_iterator != NULL) {
            if (node_iterator->data == val)
                return node_iterator->rep;
            node_iterator = node_iterator->next;
        }
        conn_comp_iterator = conn_comp_iterator->next;
    }
    return NULL;
}
```

CODE FOR IMPLEMENTATION (contd.)

```
void LINK(conn_comp *x, conn_comp *y) {
    if (x != y) {
        conn_comp *max, *min;
        if (x->len > y->len) {
            max = x;
            min = y;
        }
        else {
            max = y;
            min = x;
        }
        max->len += min->len;
        node *changin_pt = min->head;
        max->tail->next = min->head;
        max->tail = min->head;
        while (changin_pt != NULL) {
            changin_pt->rep = max;
            changin_pt = changin_pt->next;
        }
        if (min->prev != NULL) min->prev->next = min->next;
        if (min->next != NULL) min->next->prev = min->prev;
        delete min;
    }
}
```

```
void UNION(edge temp, conn_comp *start)
{
    if (temp.value1 != temp.value2)
        LINK(FIND_SET(temp.value1,start), FIND_SET(temp.value2,start));
}
```

TREE IMPLEMENTATION

- Each node in this implementation has two additional pieces of information besides its value: **rank** and **parent** pointer.
- The rank of a node indicates the size of the subtree with that node as the root, while the parent pointer points to one of the parent nodes, ideally the highest node in the tree (the main root).

IMPLEMENTATION OF OPERATIONS

- Make_Set initializes a node with its data, rank set to 1, and parent pointer pointing to itself.
- Find_Set recursively updates the parent pointers of all predecessors of a node until it reaches the head root node of the tree.
- Union makes the lower rank node a child of the higher rank node. If the ranks are equal, the parent node's rank is incremented by one. Given the number of nodes in the dataset, we can create an array of nodes of size n.

CODE FOR IMPLEMENTATION

```
void MAKE_SET(node *x) {
    x->parent = x;
    x->rank = 0;
}

node *FIND_SET(node *x) {
    if (x != x->parent)
        x->parent = FIND_SET(x->parent);

    return x->parent;
}

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

    if (x->rank == y->rank)
        y->rank++;
}

void UNION(edge temp) {
    if (temp.value1 != temp.value2)
        LINK(FIND_SET(&node_list[temp.value1]), FIND_SET(&node_list[temp.value2]));
}
```

OBSERVATIONS ON DATASETS

- SNAP Facebook (vertices: 4039, edges: 88234)
    1. Tree-based: 125.446ms
    2. Linked-list based: 134.665ms
- SNAP Journal (vertices: 4847571, edges: 68993773)
    1. Tree-based: 125175ms (2mins)
    2. Linked-list based: 1% completion in 1hr
- KONECT USA Western (vertices: 6262104, edges: 15119284)
    1. Tree-based: 24509.2ms (24secs)
    2. Linked-list based: Approximately 1hr to reach 20% completion

KRUSKAL'S MST IMPLEMENTATION

Observations for SNAP Facebook Dataset:

Time taken: 3476ms

```
void kruskalMST(vector<Edge>& edges, int V) {
    sort(edges.begin(), edges.end(), compareEdges);

    Subset* subsets = new Subset[V];
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    vector<Edge> result;
    int e = 0;
    for (Edge next_edge : edges) {
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y) {
            result.push_back(next_edge);
            Union(subsets, x, y);
            e++;
        }
    }

    delete[] subsets;

    cout << "Edges in the MST are: " << endl;
    for (Edge edge : result) {
        cout << edge.src << " - " << edge.dest << " (weight: " << edge.weight << ")" << endl;
    }
}
```

THANK YOU