# CS301- Algorithms

# Fall 2021-2022

# Project Progress Report

## 1. Problem Description

a) There will be a randomly generated undirected graph G(V, E) and the problem wants us to generate a number k which will be equal to the number of the highest degree of that undirected graph's spanning tree. The degree of a node is the number of edges that are connected to that node. We will need to look at every spanning tree's node and count the edges which are linked to that node and the highest value will be equal to k. That k will be the minimum integer which no other node will have a greater degree than it. This problem may also be seen in optimization problems especially in distance optimization problems in real life.

**Our problem is NP-Complete**: Our problem can be solved in two main steps by using the nondeterministic algorithm which follows. First, we will nondeterministically choose for every edge if it will be included in the tree. After that, we will check if our tree is still a tree and all the vertices have degrees less than the number k. To prove that our algorithm is NP-complete we need to reduce a known NP-complete problem to our spanning tree problem. As an example, we will utilize the knowledge that the problem Hamiltonian Path is an NP-complete problem. We know that a Hamiltonian path exists if and only if we can traverse the tree by only visiting a vertex once. When we consider this rule we can adapt this to our problem. Let's take the value k = 2 in our problem. By doing that we will need to have 2 as the maximum degree of our tree. If we can find such a tree we would be finding a solution to the Hamiltonian path. On the other hand, if our graph has a Hamiltonian graph in it, that means the value k will be equal to 2 and it will definitely be a spanning tree. As a result of these claims, we were able to reduce the Hamiltonian path problem to our problem. Hence we can say that our problem is NP-complete.

## 2. Algorithm Description

a) In our code, we initially create a matrix of booleans that we will use for the main part of our algorithm. We ask for the input by the user regarding the number of nodes and edges desired by the user. Regarding those inputs, there is a simple rule that determines whether a graph can be generated with the inputs given by the user. If this pair of numbers can't be used to create an appropriate undirected graph, the user will be asked to enter new inputs until it is mathematically possible to create a graph with the given input. Once we have taken care of the inputs, the size of the matrix is determined based on the number of nodes entered in the input, creating a square of boolean values that has VxV elements where V is the number of nodes, each of them initially false. The diagonal section of the matrix ([i,i] for every 0 <= i <= V) is considered false, while a random number generator inverts the boolean of E (number of edges) amount of boolean values at the top side of [i, i] to true while making sure that at least one boolean value of each row is true. This boolean value at the top side of the matrix is copied into the bottom side, where the undirected graph is complete. Then we create a spanning tree.

After we create the spanning trees, for each one of them, the values of each row are summed up to find the maximum value of connections for each node, which becomes the k value of said spanning tree. After checking and determining the k values of all the possible spanning trees, the minimum value of k is determined to be the answer of the project problem we started with, given the specific node and edge input.

**Pseudocode:**

```
Create boolean matrix called mat
Input n(node) and e(edges)
repeat
        ask for input n and e
until (e < n-1 || e > n*(n-1)/2)
Call Undirected_Fill_Graph
        Call init_matrix with matrix, node, and node, which creates the initial matrix
        For the amount of edges
                Change a random point and its symmetric to true
```

Call route_checker_FULL, that checks that every node is connected to at least one other node, ie. every line in the matrix has at least one true value, using the route_checker_line function

Call create_sub_trees

Create an integer matrix called GRAPH

Call matrix_to_graph with mat and GRAPH to define the edges of the matrix by connecting nodes according to the input

Create integer matrix called COMB_V

Call create_combination with Graph.size(), mat.size()-1 and COMB_V, as well as the go function to label the edges to be used, which will create all the possible spanning trees

Create boolean matrix called aux_mat

Call init_matrix with aux_mat and mat.size()

For amount of COMB_V.size()

Call clear_matrix with aux_mat

Make a spanning tree

Call route_checker_FULL to check if the spanning tree has any unconnected nodes

Call DEGREE function to check the degrees of each node, and find the maximum value to determine the k value of that spanning tree

Call print_matrix that prints the spanning tree

Find the minimum k value amongst all the k values of all spanning trees

Print the minimum k value as the solution of the problem

b)

Our greedy algorithm for the second part of the project consists of some minor and one very major change to our algorithm/program to speed up the progress. First of all, to shorten our best case scenario solutions all the way down to O(1), we have made a new addition to our main code, where the program will terminate and give the solution as k=2 whenever the number of edges are n(n-1)/2, for n being the input of node number. The basic explanation for this is as follows: The maximum number of edges possible when creating a graph for n nodes, is n(n-1)/2, and when every possible edge is given for the graph, the program would always be able to make a spanning tree connecting the nodes linearly, creating

a minimum spanning tree with k value of 2, which is the minimum answer the code can get for every possible graph for every input duo anyway. Our next minor addition is a program termination in the middle of forming spanning trees throughout the class functions used. Whenever the program is able to find a spanning tree with a k value of 2, the code is to stop and print out the answer to our problem as 2, since again, a spanning tree can't have a degree lower than 2.

The major change to our program consists of ignoring the matrix creation part of the individual spanning tree making, and instead creating an adjacency list, resulting in a time decrease from O(|V|^2) to O(|V| + |E|). However, since this results in a directed graph, for example 0 being connected to 1, while 1 not being connected to 0, we use this process in reverse to technically transform this list into an undirected graph list.

In our new program, the way we find the k value is also significantly different and is as follows: after we create the undirected graph, we use the Dijkstra class to create random spanning trees, which we later better them to see if the k value of that tree decreases, since even when it is a spanning tree, it might not be minimum degree spanning tree since it is randomly generated. In short, for this method, we label 3 different nodes U, V and W and check certain rules about them. Once we choose the nodes, we link U and W in the spanning tree while, and removing an edge from U. If we check the new spanning tree and the maximum degree hasn't changed, we choose different edges and nodes in time, either until k hits 2, or every combination of nodes and edges, according to the restrictions, have been checked.

New Pseudocode:

Create boolean matrix called mat

Input n(node) and e(edges)

repeat

    ask for input n and e

until (e < n-1 || e > n*(n-1)/2)

If e == n*(n-1)/2, terminate the program completely and print 2 as the answer of the problem

Call Undirected_Fill_Graph

    Call init_matrix with matrix, node, and node, which creates the initial matrix

    For the amount of edges

        Change a random point and its symmetric to true

    Call s_connectivity, that checks that every node is connected to at least one other node, ie.

    every line in the matrix has at least one true value, using the route_checker_line function

Call class DJKS to create spanning tree

Call MAKE_UNDIRECTED_LIST for initial adjacency list and spanning tree separately

    create matrix as T_list

    for every node:

        push_back node and connected nodes in list to T_list

    update list to its transpose

    for every node:

        push_back node and connected nodes in list to T_list

For the spanning tree:

    Choose nodes U, V and W

repeat until deg(U)-1 >= max(deg(V), deg(W)) and V and W are connected in adjacency list

Repeat:

    Add edge between V and W

    Remove random edge from U

    Evaluate k of the new spanning tree

    update k if the new k is smaller

until k = 2 or every constricted node combination is used

Find the minimum k value amongst all the k values of all spanning trees

Print the minimum k value as the solution of the problem

Our algorithm partially uses a randomized algorithm as a part of our solution, as the program uses randomization to create an undirected graph through the inputs given to us, to later create the spanning trees.

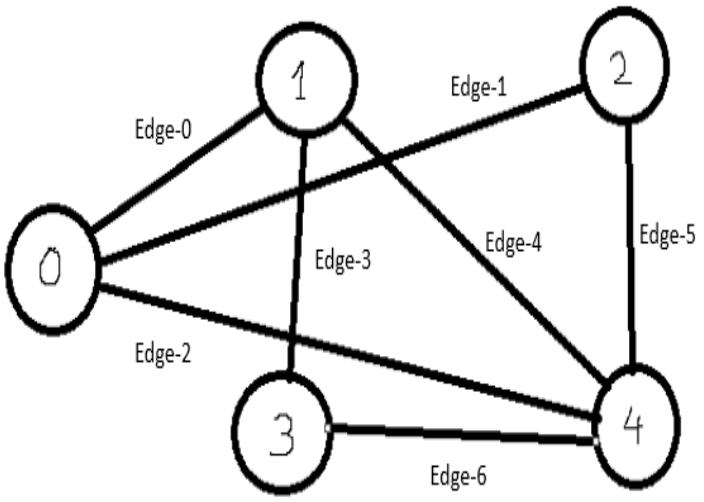
Also, our Dijkstra algorithm chooses one spanning tree and fixes it so it can have a minimum k value. However, every spanning tree has different potential for minimum degree. Some spanning tree degrees can be lowered to 2 and some cannot. Our algorithm approximates the minimum degree by picking a spanning tree and trying to decrease its degree as low as it can be. This means that we also implement approximation in our algorithm.
Our algorithm also uses greedy algorithm. If the spanning tree which was founded by Dijkstra has k value of 2, we terminate the program since k = 2 is our answer. Also if the user enters minimum number of edges that can be entered to a node, the undirected graph will be a spanning tree with k value of 2 so we terminate the program in that case as well.

## 3. Algorithm Analysis

**a)**

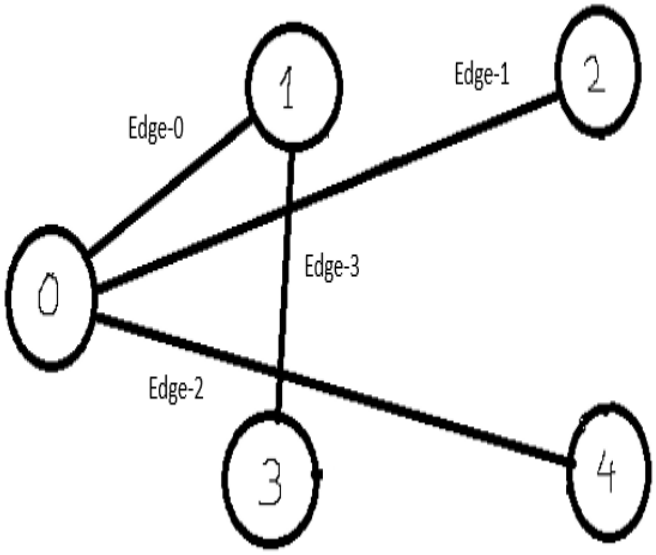
**1-** Spanning tree combines every node together with the minimum number of edges and in order to connect n nodes together we need a minimum number of n-1 edges. Let's assume that we have an undirected graph with 5 nodes and 7 edges.

```
randomized matrix
0 1 1 0 1
1 0 0 1 1
1 0 0 0 1
0 1 0 0 1
1 1 1 1 0
{0,1}
{0,2}
{0,4}
{1,3}
{1,4}
{2,4}
{3,4}
number of node: 5
number of edge: 7
number of possible s-tree: 35
```



In order for this undirected graph to be a spanning tree, we must choose 4 edges (n-1) out of 7 edges that the current example has and create subgraphs and then check if the subgraphs that we created is a spanning tree. There will be C(7,4) = 35 subgraphs to check in this case. We can choose the edges {0,1,2,3} to create a subgraph or {0,1,2,4} etc. For every combination, we create subgraphs and for each subgraph, we check if all the nodes are connected to each other and there are no multiple node groups that are not connected to each other. If the subgraph passes the test, then we count the rows of the matrix and find every node's degree, the biggest one will be the k value of that spanning tree. For example, this is a possible spanning tree subgraph of the previous undirected graph and it is created using the edges {0,1,2,3}.

```
this is spanning tree
The sum of each row: 3  2  1  1  1
##########
the k value is 3
0 1 1 0 1
1 0 0 1 0
1 0 0 0 0
0 1 0 0 0
1 0 0 0 0
{0,1}
{0,2}
{0,4}
{1,3}
number of node: 5
number of edge: 4
number of possible s-tree: 1
```




We can find the degrees of each node simply by summing up every row. For example, in this subgraph, the degree of node 0 is 3 ({0+1+1+0+1}=3} and its also the highest degree in this spanning tree so the k value is equal to 3. After we check every subgraph and calculate the k values of every spanning tree, we pick the smallest k out of every spanning tree's k value.

2-

```
void create_sub_trees(vector<vector<bool>> &mat){
    int size=(int)mat.size();
    vector<int> k_values;
    vector<vector<int>> GRAPH;
    matrix_to_graph(mat,GRAPH);
    int n_edge=(int)GRAPH.size();
    if(n_edge==size-1){
        cout<<"inital vector is spanning tree\n";
        DEGREE(mat, k_values);
        print_matrix(mat);
        return;
    }
    vector<vector<int>> COMB_V;
    create_combination(n_edge,size-1,COMB_V);
    vector<vector<bool>> aux_mat;

    init_matrix(aux_mat,size,size);
    /*****************************************/
    for (int p = 0; p < (int)COMB_V.size(); p++) {
        clear_matrix(aux_mat);
        for (int r = 0; r < (int)COMB_V[p].size(); r++) {
            int ind=COMB_V[p][r];
            int i=GRAPH[ind][0];
            int j=GRAPH[ind][1];
            aux_mat[i][j]=true;
            aux_mat[j][i]=true;
        }
        cout<<"_____\n";
        if(route_checker_FULL(aux_mat)==true){
            cout<<"this is spanning tree\n";
            DEGREE(aux_mat, k_values);
            //print_matrix(aux_mat);
        }
        else{
            cout<<"this is not a spanning tree\n";
        }
        print_matrix(aux_mat);
        vector<vector<int>> aux_GRAPH;
        matrix_to_graph(aux_mat,aux_GRAPH);

        cout<<"_____\n";
    }
    int tmp = size;
        for(int i = 0; i < (int)k_values.size(); i++){
            if (tmp > k_values[i]){
                tmp = k_values[i];
            }
        }
        cout<<"Smallest k value is: "<<tmp<<endl;
}
```

The main function that we utilize to solve the problem is the above function create_sub_trees. In it, we have different functions, and to find the complexity we also need to find the complexities of the other functions. The functions that we need to analyze are matrix_to_graph, DEGREE, print_matrix, create_combination, go,  init_matrix, clear_matrix, route_checker_LINE, route_checker_FULL.

**matrix_to_graph:**

```
void matrix_to_graph(const vector<vector<bool>> &mat,vector<vector<int>> &graph){
    graph.clear();
    graph.shrink_to_fit();
    int size=(int)mat.size();
    vector<int>EDGE(2,0);
    for(int i = 0; i < size; i++){
        for(int j = i + 1; j < size; j++){
            if(mat[i][j]){
                EDGE[0]=i;
                EDGE[1]=j;
                graph.push_back(EDGE);
            }
        }
    }
    int n_edge=(int)graph.size();
    for(int i = 0; i < n_edge; i++){std::cout<<"{"<<graph[i][0]<<","<<graph[i][1]<<"}\n";}

    int f_n=fact(n_edge);
    int f_k=fact(size-1);
    int f_nk=fact(n_edge-(size-1));
    int n_comb=f_n/(f_k*f_nk);
    cout<<"number of node: "<<size<<endl;
    cout<<"number of edge: "<<n_edge<<endl;
    cout<<"number of possible s-tree: "<<n_comb<<endl;
}
```

The above function is used for representing the matrix as an undirected graph. By doing that we are able to transform the matrix into a graph and print the information of that graph. In the function, we have nested for loops. From that part, we have O($n^2$)complexity when n is equal to size. Hence the function has O($n^2$)complexity. Also, the function has facts(x) function which is basically taking the factorial of x. It has O(n) complexity so it does not affect the overall complexity of the function matrix_to_graph.

**DEGREE**

```
160 void DEGREE (vector<vector<bool>> aux_mat, vector<int> &k_values){
161     int size=(int)aux_mat.size();
162     vector<int> totals;
163 
164     cout<<"The sum of each row: ";
165     for(int i=0; i<size; i++){
166         int sum = 0;
167         for(int c=0; c<size; c++)
168             sum += aux_mat[i][c];
169         totals.push_back(sum);
170         cout<<totals[i]<<"  ";
171 }
172 
173     int k = 0;
174     for(int i = 0; i < (int)totals.size(); i++){
175         if (k < totals[i]){
176             k = totals[i];
177         }
178     }
179 
180     //int k = *max_element(totals.begin(), totals.end());
181     k_values.push_back(k);
182     cout<<endl;
183     cout<<"##########"<<endl;
184     cout <<"the k value is "<< k << endl;
185 
186 }
```

In this function, we calculate the degree k which is the smallest integer in the spanning trees. The first for loop which is *for(int i=0;  i<size; i++)* , we get O(n) time complexity for n=size. After that, the second loop which is *for(int c=0; c<size;c++)*, we get again O(n) time complexity.  Since previously mentioned for loops are connected, the time complexity for them will be O($n^2$). The third for loop which is *for(int i=0; i<(int)totals.size(); i++)* has smaller time complexity than O($n^2$). For this reason, the first two for loops' time complexity has been calculated. Totally, we get O($n^2$) time complexity for n = number of nodes.

**print_matrix:**

```
11  void print_matrix(vector<vector<bool>> &mat){
12      int nrows=(int)mat.size();
13      int ncols=(int)mat[0].size();
14      for(int i=0;i<nrows;i++){
15          cout<<mat[i][0];   //the first character must be printed out of the loop
16                             // to avoid space char at the end of the line
17          for(int j=1;j<ncols;j++){
18              cout<<" "<<mat[i][j];  //put space between every character, line is filled
19          }
20          cout<<endl;  //go to the next line
21      }
22  }
```

In this function, we create a matrix. The first for loop which is *for(int i=0;  i<nrows; i++)*, we get O(n) time complexity for n = number of rows. After that, the second loop which is *for(int i=0; i<ncols;c++)*, we get again O(n) time complexity for n = number of columns. These for loops are connected and finally, we get  $O(|V|\text{^}2)$ ) time complexity for V=vertices.

**create_combination:**

```
void create_combination(const int &n, const int &k,vector<vector<int>> &COMB_V){
    vector<int> combination;
    vector<int> indx;
    for (int i = 0; i < n; i++) { indx.push_back(i);}
    go(0, k,indx,combination,COMB_V);
    for (int i = 0; i < (int)COMB_V.size(); i++) {
        cout<<"{"<<COMB_V[i][0];
        for (int j = 1; j < (int)COMB_V[i].size(); j++) {
            cout<<","<<COMB_V[i][j];
        }
        cout<<"}\n";
    }
}
```

The create combination function will mainly focus on finding the combinations possible and giving us them. Inside create_combination we also have another function which is the go function. First, we need to analyze it.

**go:**

```
void go(int offset, int k,vector<int> &indx,vector<int> &combination,vector<vector<int>>& COMB_V) {
    if (k == 0) {
        COMB_V.push_back(combination);
        return;
    }
    for (int i = offset; i <= (int)indx.size() - k; ++i) {  T(k) = (n-k) T(k-1)
        combination.push_back(indx[i]);
        go(i+1, k-1,indx,combination,COMB_V);
        combination.pop_back();
    }
}
```

The go function does the hard work for the create_combination function. It works recursively and to analyze its complexity we need to write:

It basically generates all possible combinations needed in our algorithm. Since every element will be either an element of the combination or not we will need to make the choice accordingly. While iterating over the values we will be pushing or popping a single element

out of the combinations vector and therefore we will be finding all the possible combinations. This process will result in $\Theta(2^n)$ time complexity.

Now we can continue analyzing the create_combination function. We see that the go function is being called which results in $\Theta(2^n)$ complexity. When we continue we see two nested for loops both of them iterating n (size) times. From that part, we have $O(n^2)$complexity. To conclude we can say that $\Theta(2^n)$ from the go function will dominate and create_combination will have $\Theta(2^n)$ time complexity.

**DFSUtil**

```
/*************************************************************/
//O(V+E) fills true if vertice is reachable
void DFSUtil(const int &v, vector<bool> &visited,const vector<vector<int>> &list){
    visited[v] = true;
    for(int i = 0; i < list[v].size(); i++){
        int ind=list[v][i];
        if(!visited[ind]){DFSUtil(ind, visited,list);}
    }
}
/*************************************************************/
```

This function fills true if vertex is reachable. Time complexity of this function is O(V+E).

**clear_matrix:**

```
24 void clear_matrix(vector<vector<bool>> &mat){
25     int nrows=(int)mat.size();
26     int ncols=(int)mat[0].size();
27     for(int i=0;i<nrows;i++){
28         for(int j=0;j<ncols;j++){mat[i][j]=false;}
29     }
30 }
```

In this function, we clear the matrix. Similarly, with the previous function, we have two for loops connected to each other. First one is *for(int i=0; i<nrows; i++)* and second one is *for(int i=0; i<ncols;c++).* These for loops have O(n) time complexity for each. Finally, we get O(|V|^2) time complexity for V=vertices because these for loops are connected.

**init_matrix:**

```
32  void init_matrix(vector<vector<bool>> &mat,const int &nrows,const int &ncols){
33      mat.resize(nrows);              // row size is determined
34      for(int i=0;i<nrows;i++){
35          mat[i].resize(ncols);     // column size is determined for each row
36      }
37      clear_matrix(mat);
38  }
```

In this function, we create the initial matrix as an empty matrix. We have one for loop which is *for(int i=0;  i<nrows; i++)* and we know that its time complexity is O(n). However, we call a function which is named clear_matrix(mat). For this reason, line 37 has  $O(n^2)$ time complexity. Totally, we have O(n) + $O(n^2)$ time complexity which equals O(|V|^2) for V=vertices.

**route_checker_FULL and route_checker_LINE**

```
bool route_checker_LINE(const vector<vector<bool>> &mat, const int &id){
    /* Returns false if there is no connection from the node given with id to the all oothers */
    int size=(int)mat.size();
    bool changed=false;
    vector<bool>  bROUTE(size,false);
    for(int i=0;i<size;i++){bROUTE[i]=(mat[id][i]==1);}
    do{
        changed=false;
        for(int i=0;i<size;i++){
            if(bROUTE[i]==true){
                for(int j=0;j<size;j++){
                    if(bROUTE[j]==false && mat[i][j]==true){
                        changed=true;
                        bROUTE[j]=true;
                    }
                }
            }
        }
    }while(changed==true);
    //for(int i=0;i<size;i++){cout<<bROUTE[i]<<"\t";}
    //cout<<endl;
    for(int i=0;i<size;i++){if(bROUTE[i]==false){return false;}}
    return true;
}
/*************************************************************/
bool route_checker_FULL(const vector<vector<bool>> &mat){
    int size=(int)mat.size();
    for(int i=0;i<size;i++){
        if(route_checker_LINE(mat,i)==false){return false;}
    }
    return true;
}
```

route_checker_FULL calls route_checker_LINE n (size) times. So we need to find the complexity of route_checker_LINE. route_checker_LINE checks if there exists any connection from the node given with id to all others. route_checker_FULL performs this check for every node we have. In route_checker_LINE we have nested for loops. We have two of them. All of them iterate n (size) times. which gives a complexity of O($n^2$) for the route_checker_LINE. Fınally when we look at route_checker_FULL we see that route_checker_LINE is called n (size) times. Hence, we have O($n^3$) complexity for route_checker_FULL.

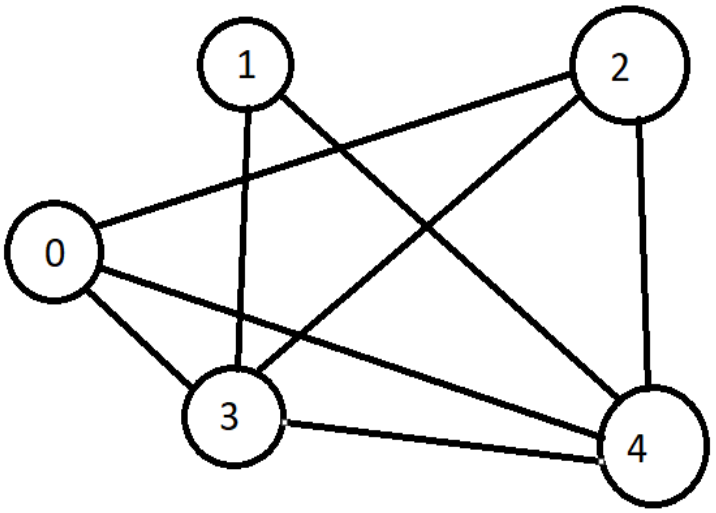
We have found the complexities of every function above. Now we are going to add them up to finally find the total complexity of our algorithm. We have 2*O($n^2$) from matrix_to_graph, 2*O($n^2$) for DEGREE, 2*O($n^2$) for print_matrix, O($2^n$) for create_combination, O($n^2$) for init_matrix, O($n^2$)for clear_matrix, and O($n^3$) for route_checker_FULL. When we combine them all we have 2*O($n^2$) + 2*O($n^2$) + 2*O($n^2$) + O($2^n$) + O($n^2$) + O($n^2$) + O($n^3$) which gives us a total time complexity of O($2^n$) for the algorithm.

**3-** When we consider the data types of our program we can say that mostly we are using vectors and especially matrices. In our code, we have integer matrices and bool matrices to perform the necessary operations of our program. When we want to consider the space complexity of our algorithm we can say that the space complexity of our algorithm is O( $n^2$) since the matrices take up $n^2$space and they will dominate the space complexity which will occur as a result of the vectors which have only O(n) space complexity.

**b)**

**1-** Our code first generates an undirected graph matrix and then stores the graph in adjacency list.

```
0 0 1 1 1
0 0 0 1 1
1 0 0 1 1
1 1 1 0 1
1 1 1 1 0


number of node: 5
number of edge: 8
number of possible s-tree: 70


FULL SCAN started

SEARCH ALGORITHM started
Graph to be checked (G)
0 {  -> 2 -> 3 -> 4 }
1 {  -> 3 -> 4 }
2 {  -> 0 -> 3 -> 4 }
3 {  -> 0 -> 1 -> 2 -> 4 }
4 {  -> 0 -> 1 -> 2 -> 3 }
```




After we get our undirected graph adjacency list, we use Dijkstra on it. (Dijkstra chooses vetrice 0 as starting point) Dijkstra algorithm returns us a spanning tree (we call it candidate graph T) and we choose 3 vertices for our algorithm to work. We choose vertice zero as U and also choose 2 other nodes as V and W. There are some rules for choosing V and W:

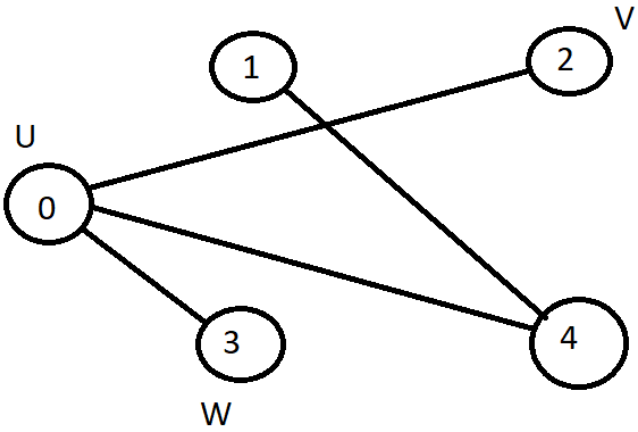
-V and W cannot be the same vertice

- In the global graph (The graph that we initially get at first) there should be edge between V and W

-Must satisfy the degree condition.

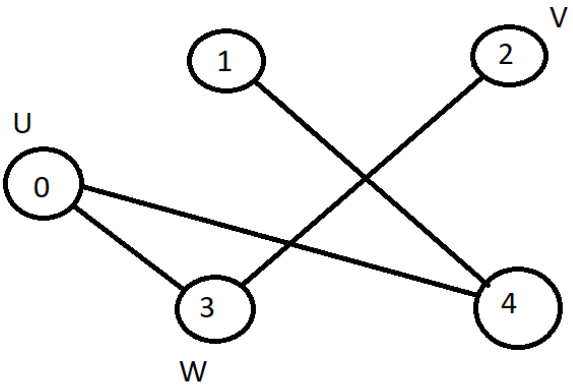
degree cond: max(degree(V), degree(W)) <= degree(U) - 2

```
Candidate graph(T)

0 {  -> 2 -> 3 -> 4 }
1 {  -> 4 }
2 {  -> 0 }
3 {  -> 0 }
4 {  -> 0 -> 1 }

_____________________________
Initial degree is: 3

0 {  -> 3 -> 4 }
1 {  -> 4 }
2 {  -> 3 }
3 {  -> 0 -> 2 }
4 {  -> 0 -> 1 }

1 optimization using falloving edges u: 0  v: 2  w: 3
new degree after reduction 2
graph has min degree spanning tree (MDST)
 terminating...
TOTAL NUMBER of IMPROVEMENTS: 2
```




After properly choosing U, V and W, we remove vertice U's first edge which is in this case the edge that combines vertice zero with 2 and add edge between V and W.

```
Candidate graph(T)

0 {  -> 2 -> 3 -> 4 }
1 {  -> 4 }
2 {  -> 0 }
3 {  -> 0 }
4 {  -> 0 -> 1 }

_____________________________
Initial degree is: 3

0 {  -> 3 -> 4 }
1 {  -> 4 }
2 {  -> 3 }
3 {  -> 0 -> 2 }
4 {  -> 0 -> 1 }

1 optimization using falloving edges u: 0  v: 2  w: 3
new degree after reduction 2
graph has min degree spanning tree (MDST)
 terminating...
TOTAL NUMBER of IMPROVEMENTS: 2
```




In this example it finds the minimum k value in single optimization but for more complex spanning trees, when we delete U's edge and combine V and W, if a vertice becomes isolated or we do not obtain a smaller degree than we already have, we choose new valid U and W vertices. After checking for each combination, we move on to another edge of U and keep choosing new V and W vertices and doing the same things until there are no vertices of U to check.

**2-** The main function that we utilize to solve the problem is the above functionL_SEARCH. We have different functions in it, and to find the complexity we also need to find the complexities of the other functions. The functions that we need to analyze are matrix_to_list, dijkstra, MAKE_UNDIRECTED_LIST, print_list, remove_edge_ind and add_edge.

# L_SEARCH

```
// O((T* + logV) * O(find_uvw)), T* = overall minimum degrees
void L_SEARCH(const vector<vector<bool>> mat){
    vector<vector<int>> G;
    vector<vector<int>> T;

    matrix_to_list(mat,G);
    dijkstra(mat, 0,T);

    MAKE_UNDIRECTED_LIST(G);
    MAKE_UNDIRECTED_LIST(T);

    cout<<"Graph to be checked (G)";
    print_list(G);
    cout<<"Candidate graph(T)\n";
    print_list(T);
    int init_degree=find_degree(T);

    cout<<"_____\n";
    if(init_degree<=2){
        cout<<"graph alreaedy has min degree spanning tree (MDST)\n";
        return;
    }else{
        cout<<"Initial degree is: "<<init_degree<<"\n";
    }
    /*********************************************************************************/
    int u=0,v=0,w=0;
    int degree=init_degree;
    int old_degree=init_degree;
    int cnt = 0;
    int pass=0;
    while(find_uvw(G,T,u,v,w)){
        remove_edge_ind(T,u, 0);
        add_edge(T,v, w);
        print_list(T);
        degree=find_degree(T);
        pass++;

        cout<<pass<<" optimization using falloving edges u: "<<u<<"  v: "<<v<<"  w: "<<w<<"\n";
        cout<<"new degree after reduction "<<degree<<"\n";
        if(old_degree>degree){
            old_degree=degree;
            cnt=0;
            if(degree<=2){
                cout<<"graph has min degree spanning tree (MDST)\n terminating...\n";
                break;
            }
        }else{
            if(old_degree<degree){
                cout<<"getting worst terminating...\n";
                break;
            }
            cnt++;
            cout<<"number of iteration without any improvement is "<<cnt<<endl;
            if(cnt>(int)T.size()){
                cout<<"terminating...\n";
                break;
            }
        }
        cout<<"_____\n";
    }
    cout<<"TOTAL NUMBER of IMPROVEMENTS: "<<degree<<"\n";
    cout<<" k value is "<<degree<<endl;
    //return degree;
}
/*********************************************************************/
```

**print_info**

```
//O(V^2) because the adcacency matrix is symmertirc, tightest bound is V^2/2
void print_info(const vector<vector<bool>> &mat){
   int V=(int)mat.size();
   int E=0;
   for(int i = 0; i < V; i++){
      for(int j = i + 1; j < V; j++){
         if(mat[i][j]){E++;}
      }
   }
   cout<<"\n";
   cout<<"number of node: "<<V<<"\n";
   cout<<"number of edge: "<<E<<"\n";
   //cout<<"number of possible s-tree: "<<fact(E)/(fact(V-1)*fact(E-(V-1)))<<"\n";
   cout<<"number of possible s-tree: "<<ncr(E,V-1)<<"\n";
   cout<<"\n";
}
```

This function prints the information like number of nodes, number of edges and number of possible spanning trees. Time complexity of this algorithm is O(|V|^2) for V=vertices. because the adjacency matrix is symmetric. Also the tightest bound is O(|V|^2/2) for V=vertices.

**print_list**

```
/*************************************************************/
// printing adjacency list, complexity O(|V|+|E|)
void print_list(const vector<vector<int>> &list){
    int V=(int)list.size();
    cout<<"\n";
    for(int i = 0; i < V; i++){
        cout<<i<<" { ";
        for(int j = 0; j < (int)list[i].size(); j++){
            cout<<" -> "<<list[i][j];
        }
        cout<<" }\n";
    }
    cout<<"\n";
}
/*************************************************************/
```

This function prints out the adjacency list that we have created. Since storing an undirected graph into adjacency list takes O(|V|+E) time, printing the list also takes O(|V|+E) complexity.

**print_matrix**

```
// complexity O(IVI^2)  V = number of nodes (vertex)
void print_matrix(const vector<vector<bool>> &mat){
   int nrows=(int)mat.size();
   int ncols=(int)mat[0].size();
   cout<<"\n";
   for(int i=0;i<nrows;i++){
      cout<<mat[i][0];   //the first character must be printed out of the loop
                         // to avoid space char at the end of the line
      for(int j=1;j<ncols;j++){
         cout<<" "<<mat[i][j];  //put space between every character, line is filled
      }
      cout<<"\n";  //go to the next line
   }
   cout<<"\n";
}
```

In this function, we create a matrix. In the first for loop, we get O(n) time complexity for n = number of rows. We get again O(n) time complexity for n = number of columns for the second loop. These for loops are connected and finally, we get $O(|V|^2)$ time complexity for V=vertices.

**ncr**

```
const long int a = 1000000000;
long long int ncr(int n,int r) {
 long long int fac1 = 1,fac2=1,fac;
 for(int i=r;i>=1;i--,n--)
   {
      fac1 = fac1 * n;
      if(fac1%i==0)
         fac1 = fac1/i;
      else
         fac2 = fac2 * i;
   }
 fac = fac1/fac2;
 return fac%a;
}
```

This function helps us to find the factorials of the numbers. We use this function to calculate the number of possible sub-trees that we can create. We calculate it by taking combination of edge and vertice-1. However, this number can be very long so in order to show it, we used long long integer but in some cases even that is not enough.

**matrix_to_list**

```
/*************************************************************/
//O(|V|^2)
void matrix_to_list(const vector<vector<bool>> &mat,vector<vector<int>> &list){
    list.clear();
    list.shrink_to_fit();
    int V=(int)mat.size();
    vector<int> link;
    for(int i = 0; i < V; i++){
        for(int j = i + 1; j < V; j++){
            if(mat[i][j]){
                link.push_back(j);
            }
        }
        list.push_back(link);
        link.clear();
    }
}
/*************************************************************/
```

This function takes the matrix and puts it in the list adjacency form for better complexity. Also, there are two connected for loops and this function's time complexity is O(|V|^2) for V=vertices.

**edge_to_list**

```
/*************************************************************/
//O(|E|), When edge representation is used, we don't have any knowledge about number of vertices which doesn't have any connection
void edge_to_list(const vector<vector<int>> &edge,vector<vector<int>> &list){
    list.clear();
    list.shrink_to_fit();
    int E=(int)edge.size();
    int V=0;
    // find the maximum indice.
    for(int i = 0; i < E; i++){
        if(V<edge[i][0]){V=edge[i][0];}
        if(V<edge[i][1]){V=edge[i][1];}
    }
    //number of vertices = maximum indice + 1
    list.resize((size_t)V+1);
    for(int i = 0; i < E; i++){
        int ind=edge[i][0];
        list[ind].push_back(edge[i][1]);
    }
    //in some cases, graph could have a vetrice which doesn't have any connection, this information does not exist in edge representation
    //so we could not resolve this situtation. Therefore, the list representaiton may not be true.
}
/*************************************************************/
```

This function represents edges in adjacency list form. When edge representation is used, we don't have any knowledge about the number of vertices which do not have any connection. In some cases, a graph could have a vertice which doesn't have any connection. This information does not exist in edge representation. Therefore, the list representation may not be true. The overall complexity of this function is O(E).

**Transpose_LIST**

```
/*************************************************************/
// Function that returns reverse (or transpose) of this graph.
// complexity O(|V|+|E|)
void Transpose_LIST(vector<vector<int>> &list){

    int V=(int)list.size();
    vector<vector<int>> T_list(V);
    for (int i = 0; i < V; i++){
        for(int j = 0; j < list[i].size(); j++){
            int ind=list[i][j];
            T_list[ind].push_back(i);
        }
        list[i].clear();
    }
    list.clear();
    list.shrink_to_fit();
    for (int i = 0; i < V; i++){
        list.push_back(T_list[i]);
    }
}
/*************************************************************/
```

This function returns the reverse or transpose of this graph. Edges were considered undirected in the edge_to_list function. This function allows us to show undirected edges as directed. Storing a graph into adjacency list takes O(|V|+E) time, for this reason this function has O(|V|+|E|) time complexity.

**MAKE_UNDIRECTED_LIST**

```
/*********************************************************/
// Function that returns reverse (or transpose) of this graph
// complexity O(|V|+|E|), our list notation is for directed graphs so in order to store the undirected graphs, we doubled the number of edges
//with their reverse forms
void MAKE_UNDIRECTED_LIST(vector<vector<int>> &list){

    int V=(int)list.size();
    vector<vector<int>> T_list;
    for (int i = 0; i < V; i++){
        T_list.push_back(list[i]);
    }
    Transpose_LIST(list);
    for (int i = 0; i < V; i++){
        for (int j = 0; j < T_list[i].size(); j++){
            list[i].push_back(T_list[i][j]);
        }
    }
}
/*********************************************************/
```

This function combines our directed list and reversed (transposed) list together and creates an undirected list. Our list notation is for directed graphs so in order to store the undirected graphs, we doubled the edges with their reverse forms. The overall complexity of this algorithm is O(|V|+E).

**S_Connectivity**

```
/*********************************************************/
//checks if vertice is strongly connected. We check only node 0 because if that satisfies the strong connectivity,
//there is no need to check the others because the graph is undirected
//DFSUtils (O(|V|+E)) is called in this function,
//MAKE_UNDIRECTED_LIST (O(|V|+E)) is called in this function,
//complexity of the other commands are O(|V|)
//the overall complexity of this function is O(|V|+E) including all sub-commands
bool S_Connectivity(const vector<vector<int>> &list){
    int V=(int)list.size();
    vector<vector<int>> aux_LIST;
    for(int i=0;i<V;i++){aux_LIST.push_back(list[i]);}
    MAKE_UNDIRECTED_LIST(aux_LIST);
    vector<bool>  visited(V,false);
    DFSUtil(0, visited,aux_LIST);
    for (int i = 0; i < V; i++){if (visited[i] == false){return false;}}
    return true;
}
/*********************************************************/
```

This function checks if a vertice is strongly connected. we only check vertice 0 because if that satisfies the strong connectivity, there is no need to check the others because the graph is undirected. In this function we called DFSUtils (O(|V|+E), MAKE_UNDIRECTED_LIST (O(|V)+E) and the complexity of the other commands are O(|V|). The overall complexity of this function is O(|V|+E) including all sub-commands.

**find_degree**

```
//O(V) finds the degree of the graph
int find_degree(const vector<vector<int>> &list){
    int degree=(int)list[0].size();
    for (int j = 1; j < (int)list.size(); j++) {
        if(degree<(int)list[j].size()){
            degree=(int)list[j].size();
        }
    }
    return degree;
}
```

This function finds the degree of the graph. Time complexity of this function is O(V) for V=vertices.

**DFSUtil**

```
//O(V+E) fills true if vertice is reachable
void DFSUtil(const int &v, vector<bool> &visited,const vector<vector<int>> &list){
    visited[v] = true;
    for(int i = 0; i < (int)list[v].size(); i++){
        int ind=list[v][i];
        if(!visited[ind]){DFSUtil(ind, visited,list);}
    }
}
```

This function fills true if vertex is reachable. Time complexity of this function is O(V+E).

### Undirected_Graph_Fill

```
void Undirected_Graph_Fill(vector<vector<bool>> &mat, const int &V, const int E){
   init_matrix(mat,V,V);
   vector<vector<int>> LIST;
   int n1,n2;
   do{
      LIST.clear();
      for (int i=0;i<E;i++){
         do{
            n1=rand()%V;
            n2=rand()%V;
         }while(n2==n1 || mat[n1][n2]==true);
         mat[n1][n2]=mat[n2][n1]=true;
      }
      matrix_to_list(mat,LIST);
   }while(S_Connectivity(LIST)==false);// We forced the inital graph to be strongly connected,
   otherwise it will be meaningless
}                                    // to check spanning trees
```

This function takes the number of nodes and edges as a parameter and produces an undirected graph which is randomly generated with many edges and nodes. This is a greedy complexity because it has random commands, overbound might be infinite. Because it has a rand command, the computer may never leave the while loop, it is nondeterministic. In the best situation (no repeated number is generated) "while" command never executes. The complexity is O(E) if we exclude the other function calls.

### edged

```
// returns true if vertice v and w are directly connected
// O(|V|)
bool edged(const vector<vector<int>> &list,const int &v, const int &w){
   for(int i=0;i<(int)list[v].size();i++){
      if(list[v][i]==w){return true;}
   }
   return false;
}
```

This function returns true if vertice v and w are directly connected. Time complexity of this algorithm is O(|V|).

**add_edge**

```
// adds direct edge between vertices v and w
// O(1)
void add_edge(vector<vector<int>> &list,const int &v, const int &w){
    if(!edged(list,v,w)){
        list[v].push_back(w);
        list[w].push_back(v);
    }
}
```

This function adds a direct edge between two given vertices. Time complexity of this algorithm is O(1).

**remove_edge_ver**

```
// removes direct edge between vertices v and w
// O (|V|)
void remove_edge_ver(vector<vector<int>> &list,const int &v, const int &w){

    for(int i=0;i<(int)list[v].size();i++){if(list[v][i]==w){list[v].erase(list[v].begin()+i);}}
    for(int i=0;i<(int)list[w].size();i++){if(list[w][i]==v){list[w].erase(list[w].begin()+i);}}
}
```

This function removes the direct edge between  two given vertices. Time complexity of this algorithm is O(|V|).

**remove_edge_ind**

```
// removes the xth connection of vertice u
// O (|V|), we use vector.erase command which has O(n) (In this case n = V) complexity
int remove_edge_ind(vector<vector<int>> &list,const int &u, const int &x){
    int tmp=list[u][x];
    list[u].erase(list[u].begin()+x);

    for(int i=0;i<(int)list[tmp].size();i++){
        if(list[tmp][i]==u){
            list[tmp].erase(list[tmp].begin()+i);
            return tmp;
        }
    }
    return -1;
}
```

This function removes the xth connection of vertice u. Time complexity of this algorithm is O(|V|). We use the vector.erase command which has O(n) (In this case n = V) complexity.

**find_uvw**

```
// the complexity is O(V^3 * O(DFSUtil))=O(V^4)   O(DFSUtil) = O(V+E)
bool find_uvw( const vector<vector<int>> &G,
               const vector<vector<int>> &T,
               int &u,int &v, int &w){
   int V=(int)T.size();
   vector<int>d(V);
   vector<vector<int>> aux_T;
   vector<vector<int>> aux_G;
   for(int i=0;i<(int)T.size();i++){aux_T.push_back(T[i]);}
   for(int i=0;i<(int)G.size();i++){aux_G.push_back(G[i]);}

   for (int i=0;i<V;i++){
      d[i]=(int)aux_T[i].size();
      if(d[u]<d[i]){u=i;}       // u is the vertice that has max. degree
   }
   if(d[u]<=2){return false;}  // best case is achieved

   // u,v and w should not be same vertice
   for(int x_ind=0;v<(int)aux_T[u].size();v++){
      for(v=0;v<V;v++){
         if(v!=u){
            for(w=v+1;w<V;w++){// v != w is guaranteed
               // in the raw graph connection between v and w must exist
               // in the MDST graph connection between v and w must NOT exist
               if((w!=u) && MAX(d[v],d[w])<=(d[u]-1) && !edged(aux_T,v,w) && edged(aux_G,v,w)){
                  int x=remove_edge_ind(aux_T,u, x_ind);    // removes x_ind'th connection of vertice u. "0" is an arbitrary selection
                  add_edge(aux_T,v, w);
                  /***********************************/
                  // check if new form is still spanning tree
                  vector<bool>  visited(V,false);
                  DFSUtil(0, visited,aux_T);
                  bool OK=true;
                  for (int i = 0; i < V; i++){if (visited[i] == false){OK=false; break;}}
                  visited.clear();
                  if(OK){return true;}
                  /***********************************/
                  remove_edge_ver(aux_T,v,w);
                  add_edge(aux_T,u, x);
               }
            }
         }
      }
   }
   cout<<"Could not find proper u,v,w\n";
   return false; // no more improvement
}
/*****************************************************************/
```

This function finds the proper u, v and w values by checking the rules. The complexity of this algorithm is O(V^3*O(DFSUtil)). As we mentioned before, the time complexity of the DFSUtil function is  O(V+E). Overall, we get O(V^4) time complexity for this algorithm.

**minDistance**

```
int minDistance(vector<int> dist, vector<bool>sptSet){
   int min = INT_MAX, min_index=0;
   for (int v = 0; v < (int)dist.size(); v++){
      if (sptSet[v] == false && dist[v] <= min) { min = dist[v], min_index = v; }
   }
   return min_index;
}
```

This function finds the shortest distance between two nodes. Also, this function is called by the Dijkstra function. Time complexity of this algorithm is O(|V|).

**dijkstra**

```
void dijkstra(vector<vector<bool>> mat, int src,vector<vector<int>> &list){
   int V=(int)mat.size();
   vector<int> rota(V);
   rota[src] = src;

   vector<int> dist(V,INT_MAX);            // The output array.  dist[i] will hold the shortest distance from src to i
   vector<bool>sptSet(V,false);            // sptSet[i] will be true if vertex i is included in shortest
                                           // path tree or shortest distance from src to i is finalized
                                           // Initialize all distances as INFINITE and stpSet[] as false
   dist[src] = 0;                          // Distance of source vertex from itself is always 0
   for (int count = 0; count < V - 1; count++) { // Find shortest path for all vertices
      int u = minDistance(dist, sptSet);       // Pick the minimum distance vertex from the set of vertices not  yet processed.
      u is always equal to src in the first iteration.
      sptSet[u] = true;                    // Mark the picked vertex as processed
      for (int v = 0; v < V; v++){         // Update dist value of the adjacent vertices of the picked vertex.
         if (!sptSet[v] && mat[u][v] && dist[u] != INT_MAX // Update dist[v] only if is not in sptSet, there is an edge from
            && dist[u] + 1 < dist[v]) {       // u to v, and total weight of path from src to  v through u is smaller than current
            value of dist[v]
            dist[v] = dist[u] + 1;
            rota[v]=u;
         }
      }
   }

   vector<vector<int>> edge;
   vector<int>EDGE(2,0);
   for(int i = 1; i < V; i++){
      EDGE[0]=rota[i];
      EDGE[1]=i;
      edge.push_back(EDGE);
   }
   edge_to_list(edge,list);
}
```

This function implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation. This function was taken from the internet and the only modification is changing all the weights to 1. The aim to run this function is not to find a spanning tree, initial guess which will be subjected to optimization is done by this

function. By using Dijkstra contrary to our purpose, a maximum degree spanning tree is obtained. Time complexity of this algorithm is O(|V^2|).

We have found the complexities of every function above. Now we are going to add them up to finally find the total complexity of our algorithm. We have O(|V|^2) from matrix_to_list, O(|V|^2) for dijkstra, O(|V|+E) for MAKE_UNDIRECTED_LIST, O(|V|+E) for print_list, O(|V|) for remove_edge_ind, O(1) for add_edge. If we add them up, we get O((T*+logV)*O(find_uvw)) time complexity for T* = overall minimum degrees.

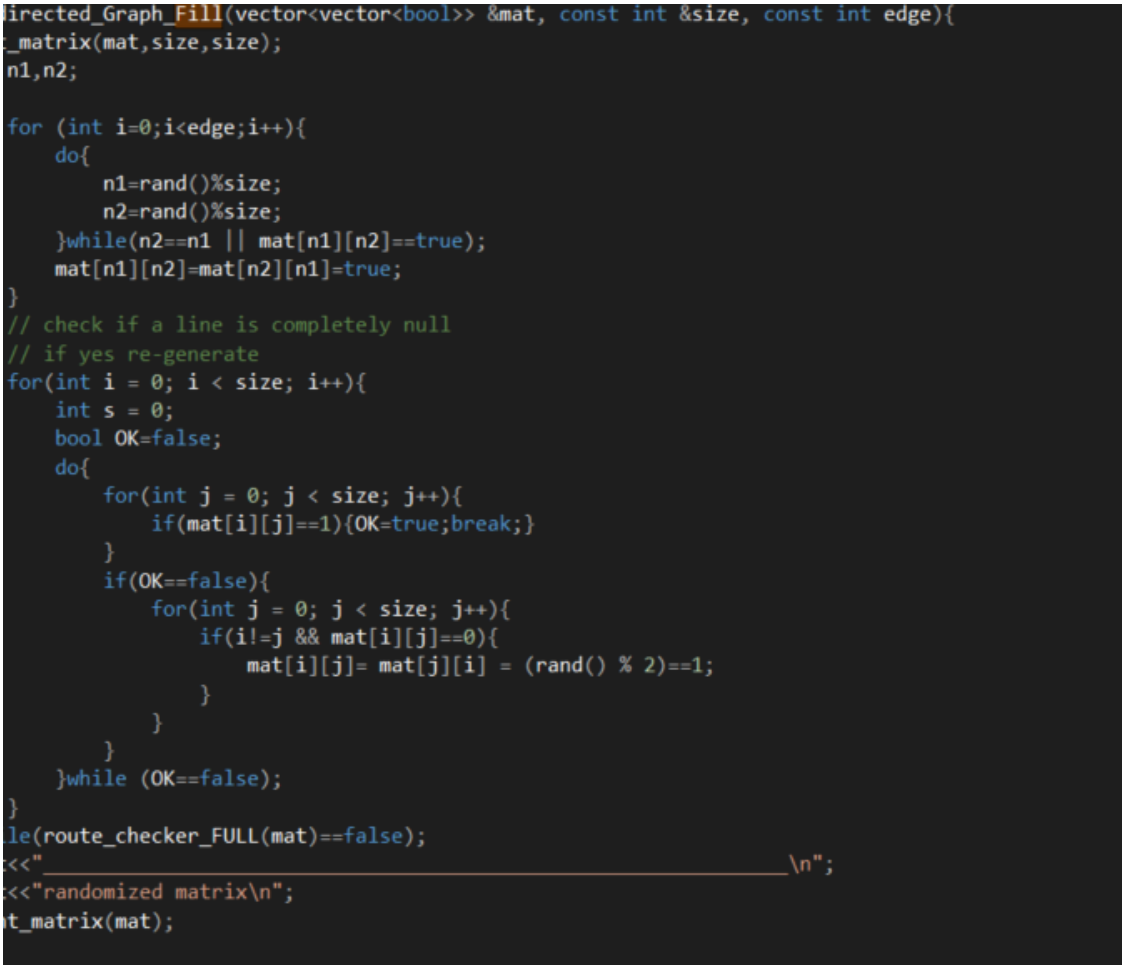
**3-** Again when we consider the data types used in our program we will see that matrices are present just like the brute force algorithm previously found. However, in our new algorithm, we have a main difference when it comes to the usage of these matrices. Rather than using matrices directly for our calculations now we use matrices for the storage purpose of our main data types which are lists. We have preferred this approach since having matrices for calculations resulted in high time complexity. When we consider the space complexity of our algorithm we can say that the space complexity of our new algorithm is again O($n^2$) since the matrices we use for storing our main lists are going to take $n^2$ space. These matrices will dominate the space complexity of the lists which only take up n space. As a result, our algorithm will have O($n^2$) space complexity (n = number of vertices).

**4. Sample Generation**

a) We have followed an approach that is taking the number of nodes and edges as input and after that creating a random undirected graph by randomizing the connection between nodes with the given number of edges. The function Undirected_Graph_Fill achieves this.

```
cout<<"How many nodes do you need? "<<endl;
cin>>n;

cout<<"how many edges do you need?"<<endl;  //minimum number of edges = n-1
cin>>e;                                     //maximum number of edges = n*(n-1)/2

while(e < n-1 || e > n*(n-1)/2){
    cout<<"not possible to create this graph, please give a different number: "<<endl;
    cin>>e;
}

cout<<"the graph will have "<<e<<" edges"<<endl;
cout<<"How many graphs do you want to generate? "<<endl;
cin>>r;

for(int i = 0; i < r; i++){
    Undirected_Graph_Fill(mat, n, e);
```

The below function Undirected_Graph_Fill takes the number of nodes and edges as a parameter and produces an undirected graph which is randomly generated with given many edges and nodes.

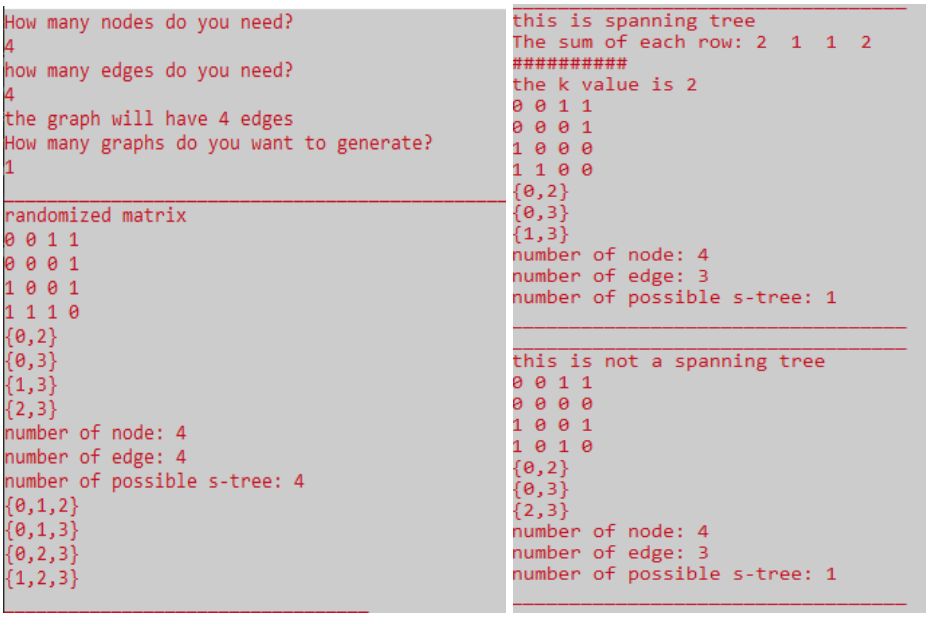


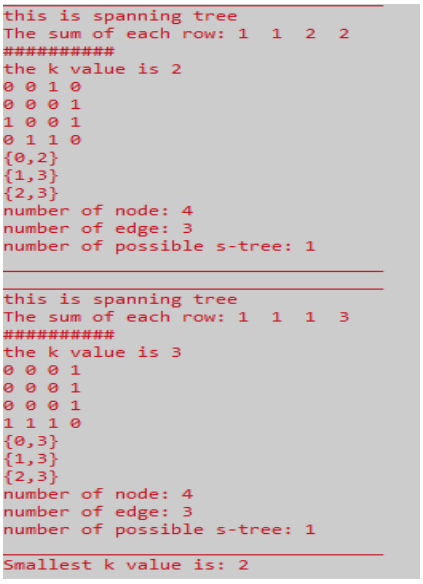
```
irected_Graph_Fill(vector<vector<bool>> &mat, const int &size, const int edge){
_matrix(mat,size,size);
n1,n2;

for (int i=0;i<edge;i++){
    do{
        n1=rand()%size;
        n2=rand()%size;
    }while(n2==n1 || mat[n1][n2]==true);
    mat[n1][n2]=mat[n2][n1]=true;
}
// check if a line is completely null
// if yes re-generate
for(int i = 0; i < size; i++){
    int s = 0;
    bool OK=false;
    do{
        for(int j = 0; j < size; j++){
            if(mat[i][j]==1){OK=true;break;}
        }
        if(OK==false){
            for(int j = 0; j < size; j++){
                if(i!=j && mat[i][j]==0){
                    mat[i][j]= mat[j][i] = (rand() % 2)==1;
                }
            }
        }
    }while (OK==false);
}
le(route_checker_FULL(mat)==false);
<<"_____\n";
<<"randomized matrix\n";
t_matrix(mat);
```

The user first enters the desired number of nodes and edges. Our code creates a undirected graph and prints out all the edges, number of nodes, number of edges and possible sup-graphs that can be created. (Our code also asks the user how many graphs should the algorithm create. The reason for this customization is that sometimes the undirected graph that we create is a spanning tree and the problem ends instantly). After we create our undirected graph, we print every possible sup-graph that can be created from that randomly generated undirected graph and check if the sup-graph is worthy enough to be a spanning tree. After we find every possible spanning tree, find the highest degree of every spanning tree and

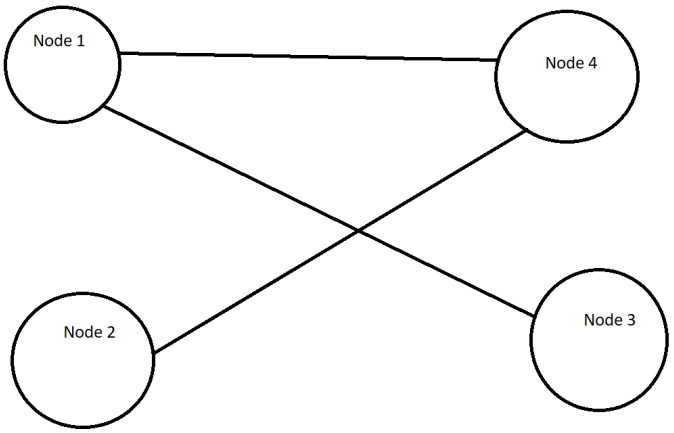
put them into a vector. At the end, we look at that vector and print out the smallest degree which will be equal to the k value.

Example-1:



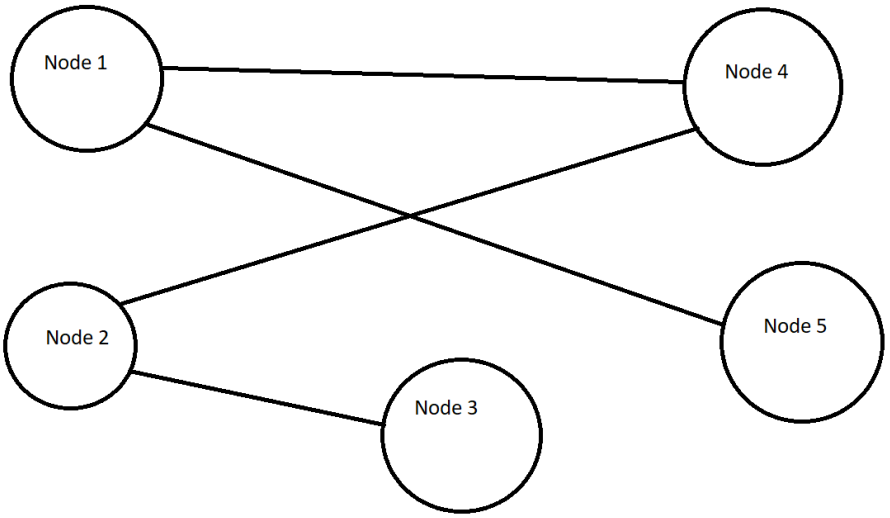

Below we can see the graph which gives us the minimum K value.




Example-2:

```
How many nodes do you need?
5
how many edges do you need?
3
not possible to create this graph, please give a different number:
4
the graph will have 4 edges
How many graphs do you want to generate?
1
_____________________________________________________________
randomized matrix
0 0 0 1 1
0 0 1 1 0
0 1 0 0 0
1 1 0 0 0
1 0 0 0 0
{0,3}
{0,4}
{1,2}
{1,3}
number of node: 5
number of edge: 4
number of possible s-tree: 1
inital vector is spanning tree
The sum of each row: 2  2  1  2  1
##########
the k value is 2
0 0 0 1 1
0 0 1 1 0
0 1 0 0 0
1 1 0 0 0
1 0 0 0 0
Press any key to continue . . .
```

Below we can see the graph which gives us the minimum K value.




Example-3:

```
How many nodes do you need?
5
how many edges do you need?
5
the graph will have 5 edges
How many graphs do you want to generate?
1
_____________________________________________
randomized matrix
0 0 1 1 1
0 0 1 0 0
1 1 0 0 1
1 0 0 0 0
1 0 1 0 0
{0,2}
{0,3}
{0,4}
{1,2}
{2,4}
number of node: 5
number of edge: 5
number of possible s-tree: 5
{0,1,2,3}
{0,1,2,4}
{0,1,3,4}
{0,2,3,4}
{1,2,3,4}
_____________________________
```

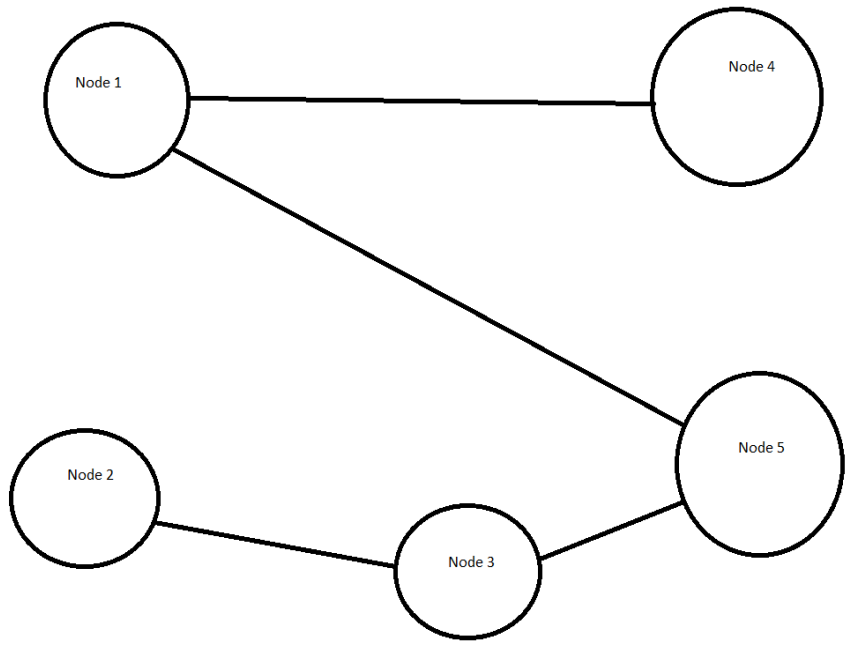
```
_____________________________________
this is spanning tree
The sum of each row: 3  1  2  1  1
##########
the k value is 3
0 0 1 1 1
0 0 1 0 0
1 1 0 0 0
1 0 0 0 0
1 0 0 0 0
{0,2}
{0,3}
{0,4}
{1,2}
number of node: 5
number of edge: 4
number of possible s-tree: 1
_____________________________________
_____________________________________
this is not a spanning tree
0 0 1 1 1
0 0 0 0 0
1 0 0 0 1
1 0 0 0 0
1 0 1 0 0
{0,2}
{0,3}
{0,4}
{2,4}
number of node: 5
number of edge: 4
number of possible s-tree: 1
_____________________________________
```

```
this is spanning tree
The sum of each row: 2  1  3  1  1
##########
the k value is 3
0 0 1 1 0
0 0 1 0 0
1 1 0 0 1
1 0 0 0 0
0 0 1 0 0
{0,2}
{0,3}
{1,2}
{2,4}
number of node: 5
number of edge: 4
number of possible s-tree: 1
```

```
this is not a spanning tree
0 0 1 0 1
0 0 1 0 0
1 1 0 0 1
0 0 0 0 0
1 0 1 0 0
{0,2}
{0,4}
{1,2}
{2,4}
number of node: 5
number of edge: 4
number of possible s-tree: 1
```

```
this is spanning tree
The sum of each row: 2  1  2  1  2
##########
the k value is 2
0 0 0 1 1
0 0 1 0 0
0 1 0 0 1
1 0 0 0 0
1 0 1 0 0
{0,3}
{0,4}
{1,2}
{2,4}
number of node: 5
number of edge: 4
number of possible s-tree: 1

Smallest k value is: 2
Press any key to continue . . .
```

Below we can see the graph which gives us the minimum K value.

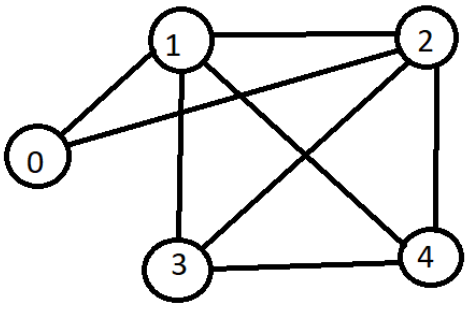


b)

Our algorithm takes the input from the user. The user enters a number of nodes and edges that the randomly generated undirected graph should have. Storing a graph in an adjacency matrix takes O(V^2) time complexity. In our previous algorithm, we created matrices for each spanning tree. Since storing a graph in a matrix takes too much time, we instead stored the graph in an adjacency list which takes O(|V|+E|) instead. We first take the matrix and put it in a list form, however, since the list form shows it as a directed graph, we take the transpose of the edges and double the number of them so the graph can be shown as an undirected list.

```
How many vertices do you need?
5
how many edges do you need?
8
the graph will have 8 edges
How many graphs do you want to generate?
1

number of node: 5
number of edge: 8
number of possible s-tree: 70
```

```
SEARCH ALGORITHM started
Graph to be checked (G)
0 {  -> 1 -> 2 }
1 {  -> 0 -> 2 -> 3 -> 4 }
2 {  -> 0 -> 1 -> 3 -> 4 }
3 {  -> 1 -> 2 -> 4 }
4 {  -> 1 -> 2 -> 3 }
Initial graph(T)

0 {  -> 1 -> 2 }
1 {  -> 0 }
2 {  -> 0 -> 3 -> 4 }
3 {  -> 2 }
4 {  -> 2 }
Initial degree is: 3
_____________________
1 optimization using falloving edges u_ind: 0  u: 2  v: 1  w: 3
new degree after reduction 2
graph has min degree spanning tree (MDST)
```




**The Pseudocode**

Create boolean matrix called mat

Input n(node) and e(edges)

repeat

ask for input n and e

until (e < n-1 || e > n*(n-1)/2)

If e == n*(n-1)/2, terminate the program completely and print 2 as the answer of the problem

Call Undirected_Fill_Graph

Call init_matrix with matrix, node, and node, which creates the initial matrix

For the amount of edges

Change a random point and its symmetric to true

Call s_connectivity, that checks that every node is connected to at least one other node, ie. every line in the matrix has at least one true value, using the route_checker_line function

Call MAKE_UNDIRECTED_LIST for initial adjacency list and spanning tree separately

create matrix as T_list

for every node:

push_back node and connected nodes in list to T_list

update list to its transpose

for every node:

push_back node and connected nodes in list to T_list

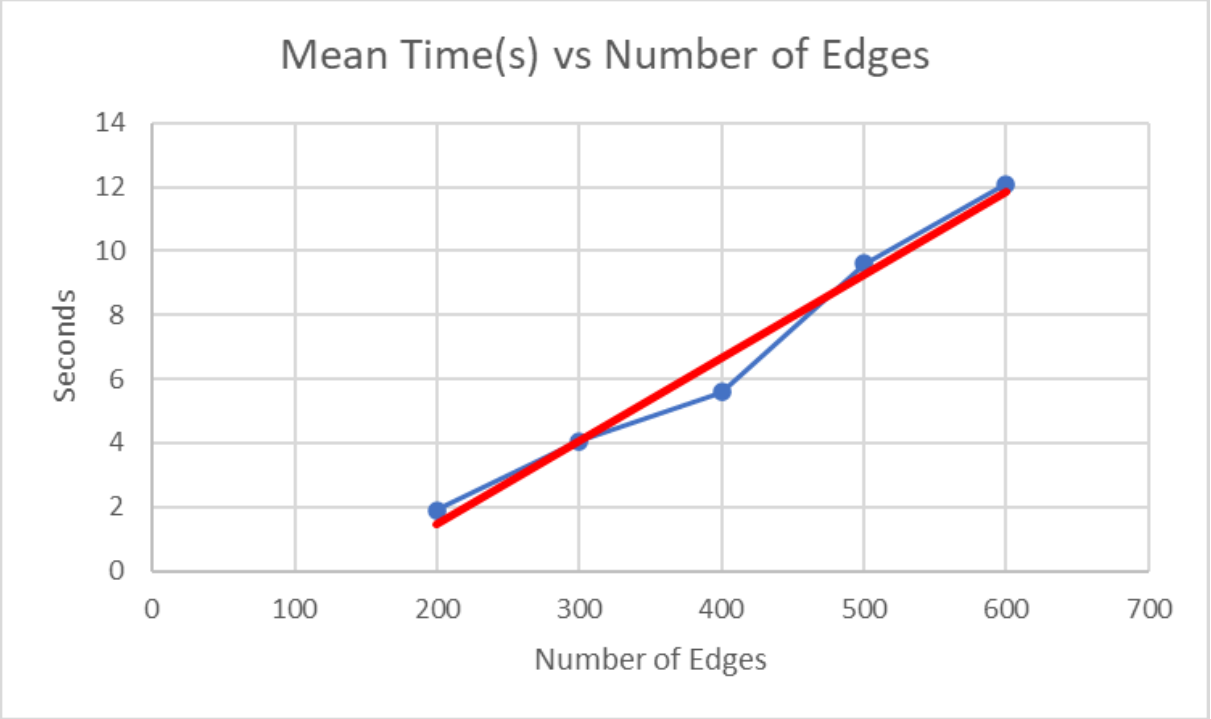
## 5. Experimental Analysis of The Performance (Performance Testing)

In order to test the performance of our algorithm, we focused on some statistical tests. We calculated the mean time, standard deviation, standard error, 90%, and 95% confidence intervals of our algorithm. We calculated these values using Excel after we extracted the time data of our algorithm. While we are conducting our experimental analysis we change the number of nodes, the number of edges, and the number of iterations. In the first part of our analysis, our aim is to check if the theoretical running time of the algorithm is seen in practice. We always had iterations bigger than 30 to Central Limit Theorem to apply. To make that check we had some cases. The first test was made 5 times with 100 nodes 100 iterations every time with an increasing number of edges from 200 to 600. All the output can be seen in the below table.

### 100 Node 100 iterations

| edge | mean Time | Standard dev. | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 200 | 1.90733139 | 0.579961524 | 0.057996152 | 0.0953951816 | 0.1136703699 |
| 300 | 4.07025572 | 0.776289844 | 0.0776289844 | 0.1276883166 | 0.1521500136 |
| 400 | 5.60028034 | 0.863940895 | 0.0863940895 | 0.1421056315 | 0.1693293040 |
| 500 | 9.61231339 | 1.490037321 | 0.1490037321 | 0.2450893292 | 0.2920419485 |
| 600 | 12.08791843 | 1.578838044 | 0.1578838044 | 0.2596957484 | 0.3094465704 |

By using the above table's edge and mean time data we constructed the below running time chart. By having the below chart we can comment on the running time of the algorithm in practice. When we look at the data points we can say that the running time of the algorithm is polynomial as also the theoretical analysis suggests. The blue line represents the data points while the red line is the best fit line.



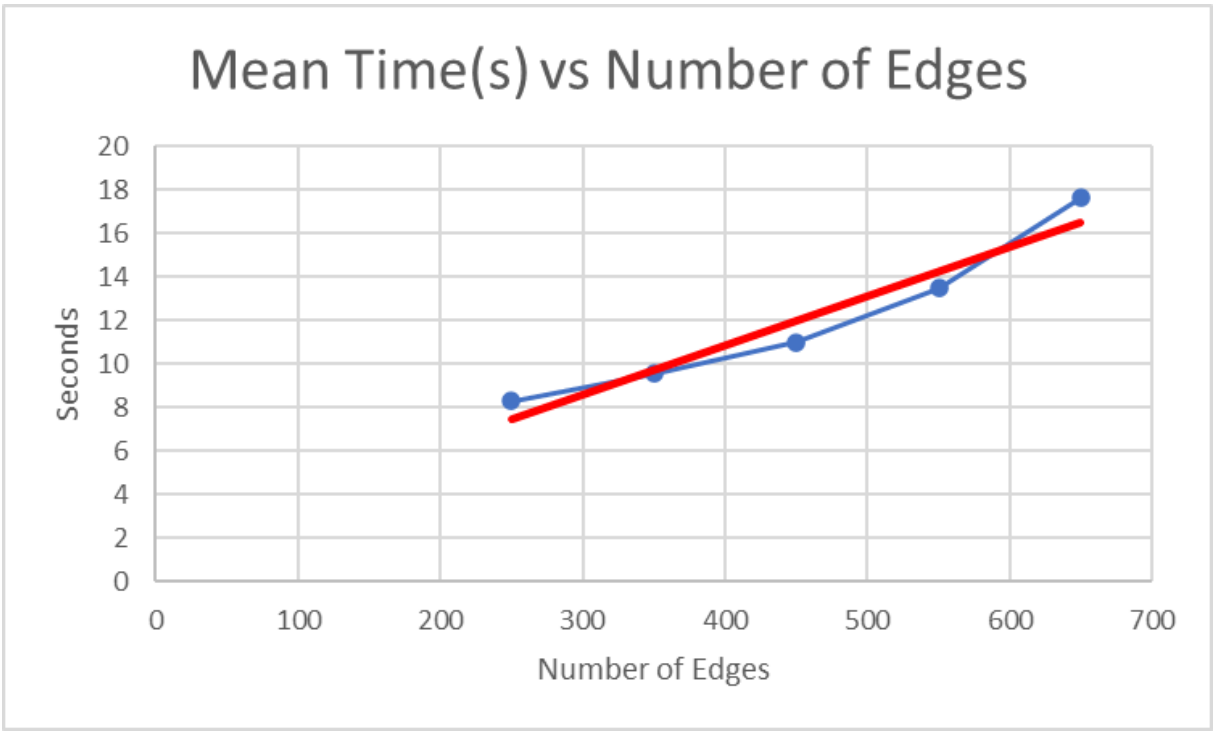

### 150 Node 50 Iterations

The second test was made 5 times with 150 nodes 50 iterations every time with an increasing number of edges from 250 to 650. All the output can be seen in the below table.

| edge | mean Time | Standard dev. | Standard Error | 90% - CL | 95% - CL |
|---|---|---|---|---|---|
| 250 | 8.27391724 | 4.285685558 | 0.428568556 | 0.839978934 | 0.7049325435 |
| 350 | 9.55862375 | 3.144662972 | 0.3144662972 | 0.6163426168 | 0.5172510294 |
| 450 | 10.98009199 | 2.538484671 | 0.2538484671 | 0.4975338531 | 0.4175435718 |
| 550 | 13.49010067 | 2.545445509 | 0.2545445509 | 0.4988981522 | 0.4186885277 |
| 650 | 17.64021922 | 6.228475279 | 0.6228475279 | 1.0244930153 | 1.2207587225 |

By using the above table's edge and mean time data we constructed the below running time chart. By having the below chart we can comment on the running time of the algorithm in practice. When we look at the data points we can say that the running time of the algorithm is again polynomial as also the theoretical analysis and previous test suggests. So we can

finally say that our algorithm has a polynomial-time complexity just like the theoretical analysis in the algorithm analysis part suggests.



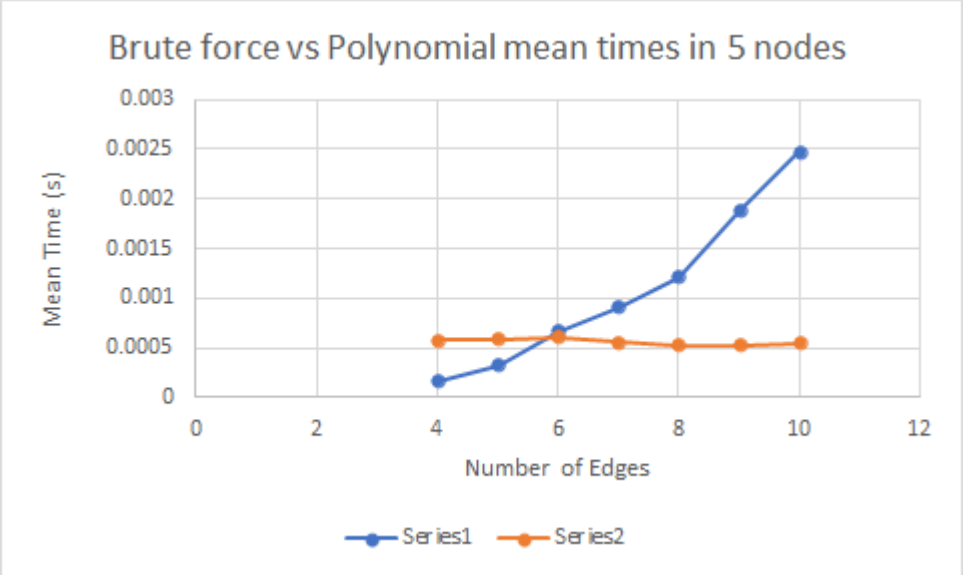

## 5 Nodes 100 Iterations

Our third test is having the number of nodes drastically reduced all the way to 5, from this point on, including the rest of the tests, we will see the true improvements our new code with p-time differs from our old brute force algorithm.

| Edge | Brute Time | P-time |
|---|---|---|
| 4 | 0.00016725 | 0.000575 |
| 5 | 0.00032869 | 0.000592 |
| 6 | 0.0006697 | 0.000607 |
| 7 | 0.0009061 | 0.000557 |
| 8 | 0.00121416 | 0.000523 |
| 9 | 0.00188153 | 0.000529 |
| 10 | 0.00247104 | 0.000545 |

For this test, since the values of the times are all much below 1, it's harder to observe the time difference between these two algorithms, but that is cleared up when we put it on the graph below, where the blue line represents brute force algorithm times, and the orange p-time.

Notice the highest value is 0.003, which indicates a very small difference for such small node values for now.

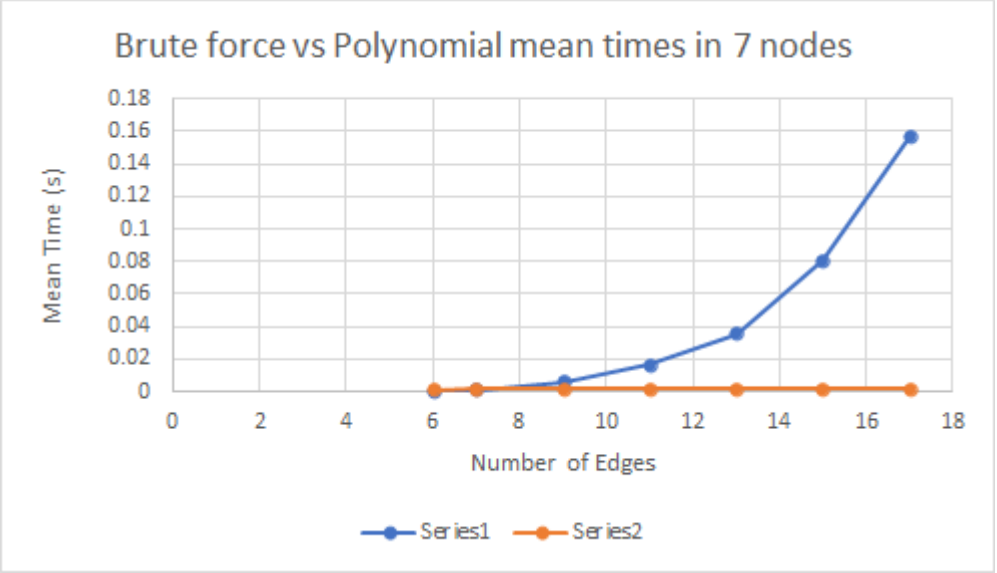



(blue line represents brute force algorithm and orange line is for representing polynomial algorithm)

## 7 Nodes 100 Iterations

| Edge | Brute Time | P-time |
|---|---|---|
| 6 | 0.00028965 | 0.001284 |
| 7 | 0.00108954 | 0.001662 |
| 9 | 0.00609534 | 0.001708 |
| 11 | 0.01615059 | 0.001805 |
| 13 | 0.0357614 | 0.001536 |
| 15 | 0.08061731 | 0.001447 |
| 17 | 0.15657287 | 0.001353 |

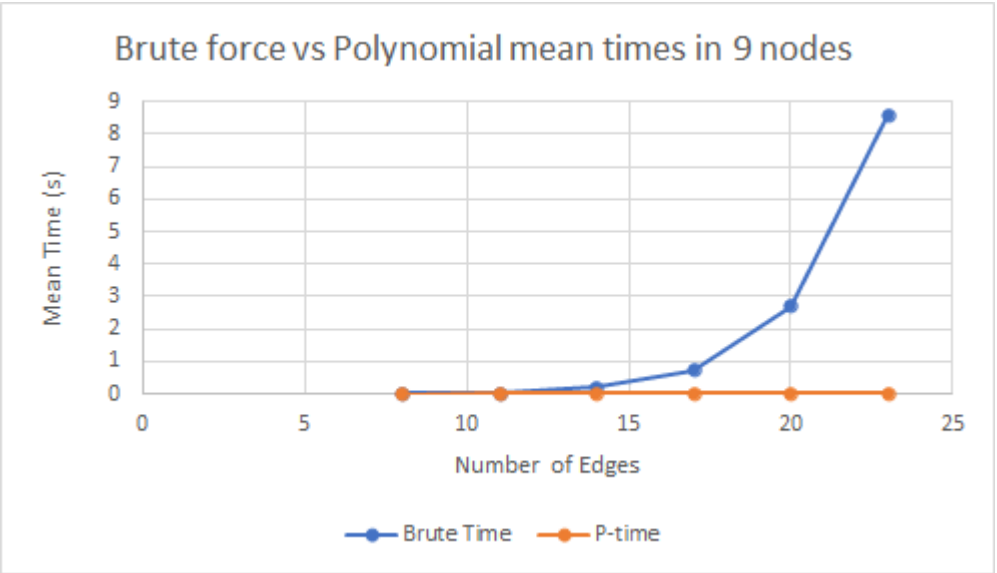
We can see the time complexity differences better when increasing the number of nodes in this test, we can see the maximum value of the graph going up to 0.18 since the longest time for brute time goes all the way up to 0.15 seconds.




## 9 Nodes 100 Iterations

| Edge | Brute Time | P-time |
|---|---|---|
| 8 | 0.00043211 | 0.002477 |
| 11 | 0.02583096 | 0.003692 |
| 14 | 0.19491531 | 0.00348 |
| 17 | 0.72802675 | 0.003843 |
| 20 | 2.69838189 | 0.004068 |
| 23 | 8.56933492 | 0.003848 |

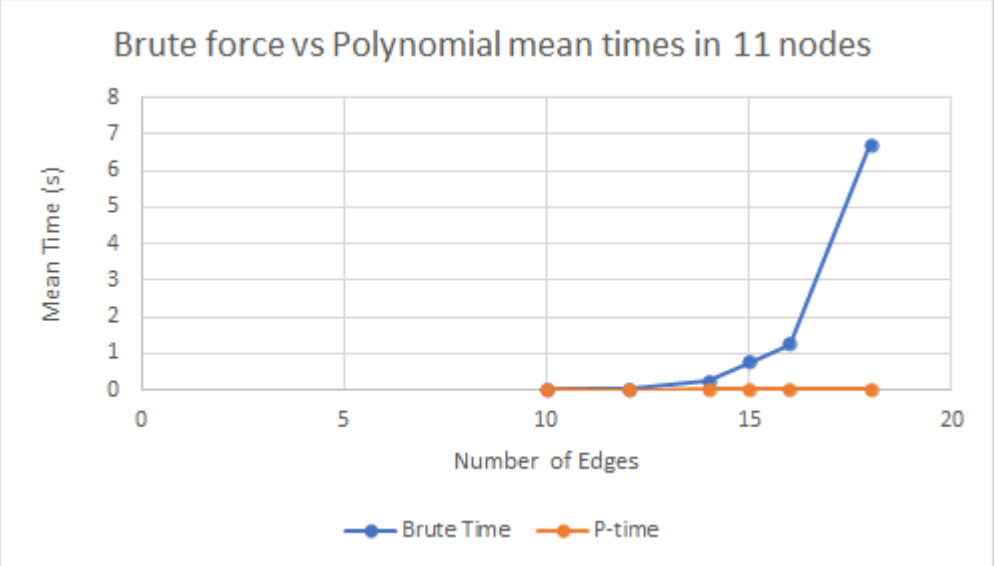
For our test with 9 nodes, the maximum time it takes for brute force time goes to 8.5 seconds while p-time stays similar, where we can observe the time difference becomes much greater.



## 11 Nodes 100 Iterations

| Edge | Brute Time | P-time |
|---|---|---|
| 10 | 0.00099679 | 0.006075 |
| 12 | 0.01761696 | 0.005931 |
| 14 | 0.22911229 | 0.007809 |
| 15 | 0.76060914 | 0.008329 |
| 16 | 1.25690999 | 0.006788 |
| 18 | 6.71543259 | 0.010736 |

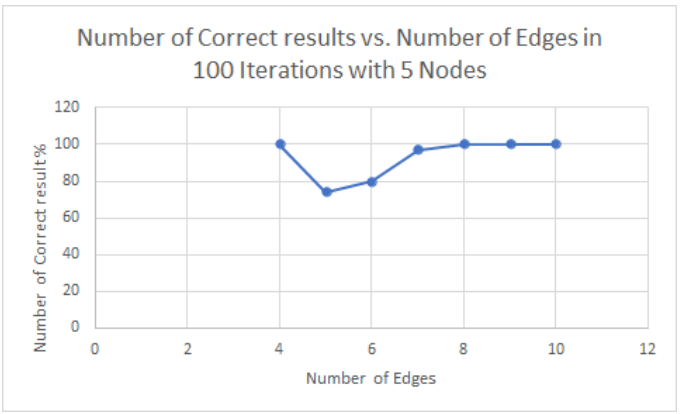
Finally, for 11 nodes, our brute force code goes only up to 6.7 seconds in here, suggesting some inconsistency. However, once we entered a higher number of edges, the brute force method was taking too long to produce even one result, taking around a minute for each tree, when we needed 100 test results, therefore, we decided not to do those tests, since the time consumed would have been a huge hindrance. Still, the p-time stays small enough to observe the time difference quite easily in the graph.



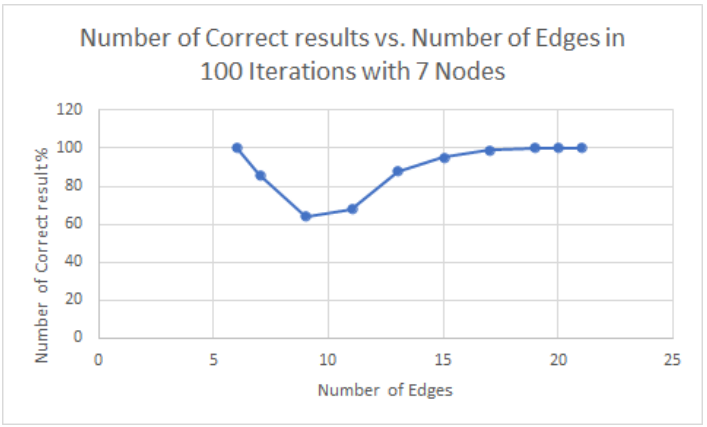
## Number of Correct Results of the New Program

We have also checked the accuracy of our algorithm based on the results from our brute force algorithm's results, since that code is designed to be always correct. Our results for 5, 7 and 9 node k-values are as follows:

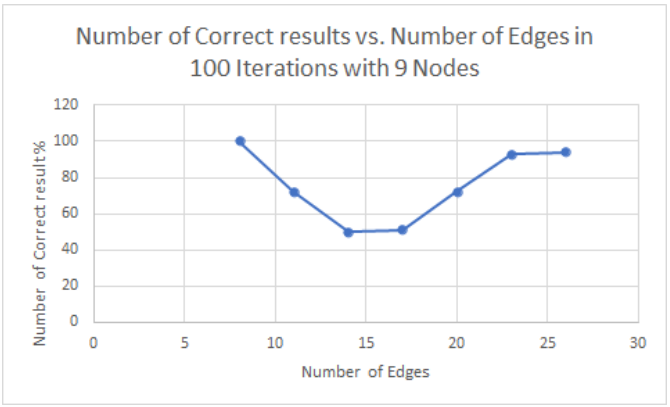
| Edge | % Correct |
|---|---|
| 4 | 100 |
| 5 | 74 |
| 6 | 80 |
| 7 | 97 |
| 8 | 100 |
| 9 | 100 |
| 10 | 100 |




| Edge | % Correct |
|---|---|
| 6 | 100 |
| 7 | 86 |
| 9 | 64 |
| 11 | 68 |
| 13 | 88 |
| 15 | 95 |
| 17 | 99 |
| 19 | 100 |
| 20 | 100 |
| 21 | 100 |




| Edge | % Correct |
|---|---|
| 8 | 100 |
| 11 | 72 |
| 14 | 50 |
| 17 | 51 |
| 20 | 72 |
| 23 | 93 |
| 26 | 94 |



We can easily see a pattern in these graphs. The smallest "number of edges" value is basically V-1, where V is the number of nodes for our input. The reason our graph starts from those values is because the graph can't be fully connected without at least that many edges. Our algorithm initially has high accuracy for the lowest edge value, while it starts decreasing in accuracy afterwards, up until V(V-1)/4. Afterwards, the rate of correct results printed from our program starts going back up, sometimes all the way up to 100% (for 5 and 7). This shows that our algorithm performs better at higher edge values in general.

**Experimental Ratio Bound**

The experimental ratio bound of our problem can be found by firstly indicating that our program has never experimentally had a mistake in assuming the true k value by more than 1. That means, for example, there has never been a case in our testing where our program gave an output of 4, when the true value is 2, or 5 and 3, etc. In that case our ratio bound is always in the form of x/(x+1) for any integer x => 2. Therefore, the farthest value our ratio bound can take from 1 results in 2/3, since every other combination of x/(x+1) where x => 2, like 3/4, 6/7, 10/11, would be closer to 1 than 2/3, which becomes our solution to this ratio bound problem.

## 6. Experimental Analysis of the Correctness (Functional Testing)

When we want to make functional testing we can use White Box testing. By giving some inputs we will try to cover all the code. As a result, we will see that our code produces an output at every possible input. Since our inputs are graphs and we mainly focus on the degrees of these graphs we will try to cover the code by giving inputs of different degreed graphs. We are going to analyze our main function which is L_SEARCH and will refer to the lines of that function from our code files which our function is from lines 96-147. Firstly we start with an initial graph that has an initial degree. In our algorithm, we first start with an initial degree check. The first case will check if that initial degree is 2 or not. As the first test case, we can have a graph that initially has a degree of two. So the following lines of the algorithm will be covered before it terminates.

Test Case 1: (a graph which has the max degree 2) [covers lines: 96 to 114]

Our second if statement is checking the relation between an old degree and a possibly optimized degree. After that check, we also have another check in order to see if the new degree is equal to 2 or not. If it is equal to 2 then it means we have again found the minimum possible degree and can terminate. So in order to fall in both of these cases, we will assume that as input we pass such a graph that after the first optimization the new degree will be smaller and it will be actually 2. This will be our second test case.

Test Case 2: (A graph which after the first optimization results with a smaller degree and that degree is 2) [covers lines: 96 to 112, 115 to 140, 143 to 146]

For a third case, we may consider input that will not directly terminate the while loop. We may input a graph that cannot have degree 2 as its maximum degree.

We will enter an input such that it will at least loop once so that it will also capture additional lines which is the desired functionality of the algorithm.

Test Case 3: (A graph which does not result with the maximum degree 2) [covers lines: 96 to 112, 115 to 132, 141,146]

After these 3 test cases, we have all the lines covered. So our white-box test is finalized. We have checked some of the important cases and saw that they are satisfied with the help of the white-box technique “statement coverage”.

## 7. Discussion

We started by constructing a brute force algorithm which was resulting in true outputs but was costing too much time. That cost of time was so big that we were not able to even finalize some test cases which were having large inputs. As a result, we constructed a greedy algorithm that had a polynomial time complexity where the brute force algorithm had exponential. These were our theoretical results but also in part 5 we had experimental analysis which also supported our claim about their time complexities. However, our polynomial-time algorithm comes with a different cost which is the low correct result rate when compared to the brute force algorithm. Again at the experimental analysis part, we saw the correct result change at our greedy algorithm. One of the most obvious problems of our algorithm is that at low edge numbers our algorithm has a bigger chance of failure but after a certain threshold our correct result rate improves and eventually becomes 100%. Even in the worst case, our rate does not fall below 50% but still, this characteristic is the biggest defect of our

algorithm. Especially at a high number of edges and nodes, our greedy algorithm produces a reliable output. We know that the high number of edges and nodes was highly costly at our brute force algorithm. So our greedy algorithm will be useful and the tradeoff between time and correct output will be profitable in our greedy algorithm. This is the main reason we construct greedy algorithms, we can finalize by saying that we have reached our goal of constructing a fast and mostly reliable algorithm.

**REFERENCES:**


**https://www.csc.kth.se/utbildning/kth/kurser/DD2354/algokomp10/Ovningar/Exercise6_Sol.pdf**

**https://www.researchgate.net/publication/220779201_Approximating_the_Minimum_Degree_Spanning_Tree_to_Within_One_from_the_Optimal_Degree**

**https://en.wikipedia.org/wiki/Graph_(abstract_data_type)**

**https://www.geeksforgeeks.org/proof-hamiltonian-path-np-complete/**