CS301 – Term Project

Project Report

Group Members:

Ece Kurnaz

Nazlı İrem Çamurlu

Tunga Berkay Savcı

Yusufcan Araz

Semester: 2020 Fall

Instructor: Hüsnü Yenigün

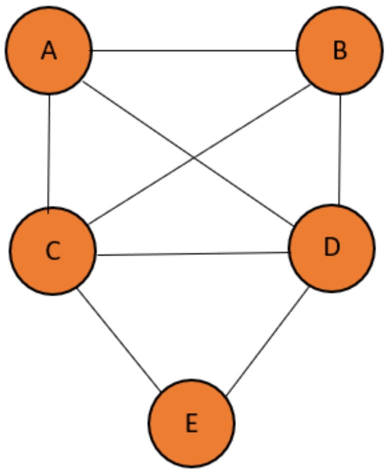
Faculty of Engineering and Natural Sciences

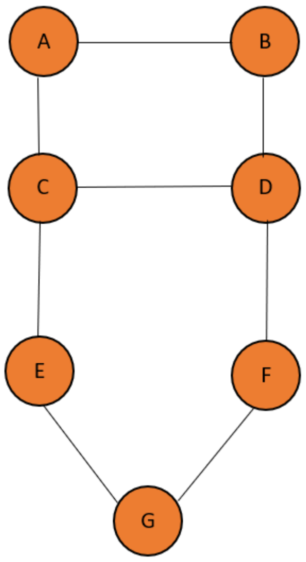
1)PROBLEM DESCRIPTION

An independent set is a subgraph of a given graph in which none of the vertices are connected with each other which would also imply that it is an anti-clique such that a clique is considered a subgraph of a given graph in which all vertices are connected with each other.

This would more formally imply that, a n-node undirected graph G(V,E) having a node set V and an edge set E and a positive integer j≤|V|. It should be considered whether there a subset V' such that V'⊆V and no two vertices in V' are joined by an edge in E.

Problem: A n-node undirected graph G(V, E) with node set V and edge set E; a positive integer k with k ≤n. Is there a subset with number of vertices W of V such that no pair of nodes in W is joined by an edge in E?





Examples of Independent Set:

{A,E} form an independent set.
{B,E} form an independent set
The largest cardinality subset of our graph is 2.

{A,D,E} form an independent set.

{B,C,F} form an independent set.
{A,D,G} form an independent set.
{B,E,F} form an independent set.
{A,E,F} form an independent set.

{A,D},{A,E},{A,F},{A,G},{B,C},{B,E},{B,F},{B,G},{D,E},{D,G},
{F,C},{C,G},{C,F} form independent sets individually.

Let's look at real-life implementations of independent sets. For example, let's say we are trying to schedule doctors in a hospital. In order to reach maximum efficiency from the doctors we cannot schedule their worktimes adjacently. For instance, if a doctor is on call for a day, we cannot schedule the same doctor on call for the other day. Therefore, we need to find the maximum independent set problem in order to find the maximum non-adjacent working hours of a doctor. The same thing applies for a security guard. If he/she is on watch for the day, we cannot schedule the same security guard for the night as it would not be safe and efficient also inhumane. Again, we could implement the maximum independent set in order to schedule the worktimes of that security guard where each vertex represents the working hours and each adjacent vertex would continue from the finishing time of the other adjacent vertex.

Independent Set is NP Complete

Proof:

Since a problem is NP Complete if it is in NP ,we need to show that;

- Independent Set is in NP.

Since all the problems in NP can be transformed into NP Complete problems NP complete problems are considered as the hardest problems in NP therefore, we need to show that;

- NP complete problem can be reduced to Independent Set in polynomial time.

a)Independent Set is in NP.

To check whether a problem is in NP, by giving a solution to the problem and an instance of the problem, which is in our case a graph G and a positive integer k, we will be able to verify our solution in polynomial time.

The solution for the independent set is a subset of V' vertices which denotes the vertices belonging our independent set. The solution can be validated through proving that no two vertex is adjacent meaning that no two vertex share an edge in between each other.

This validation, in the worst case, can be done in polynomial time, which will be $O(V^2)$;

```
Boolean flag=true;
V = {…}; //element for independent sets
for(int i=0;V.length();i++) {
    for(int j = 0; V.length();j++)
{
        if((V[i] and V[j] have an edge) && (i!=j)) //checking if there is an edge between them
{
flag = false;

Break;

}
}
}
if(flag == true)
```

{

cout<<"Solution is true";

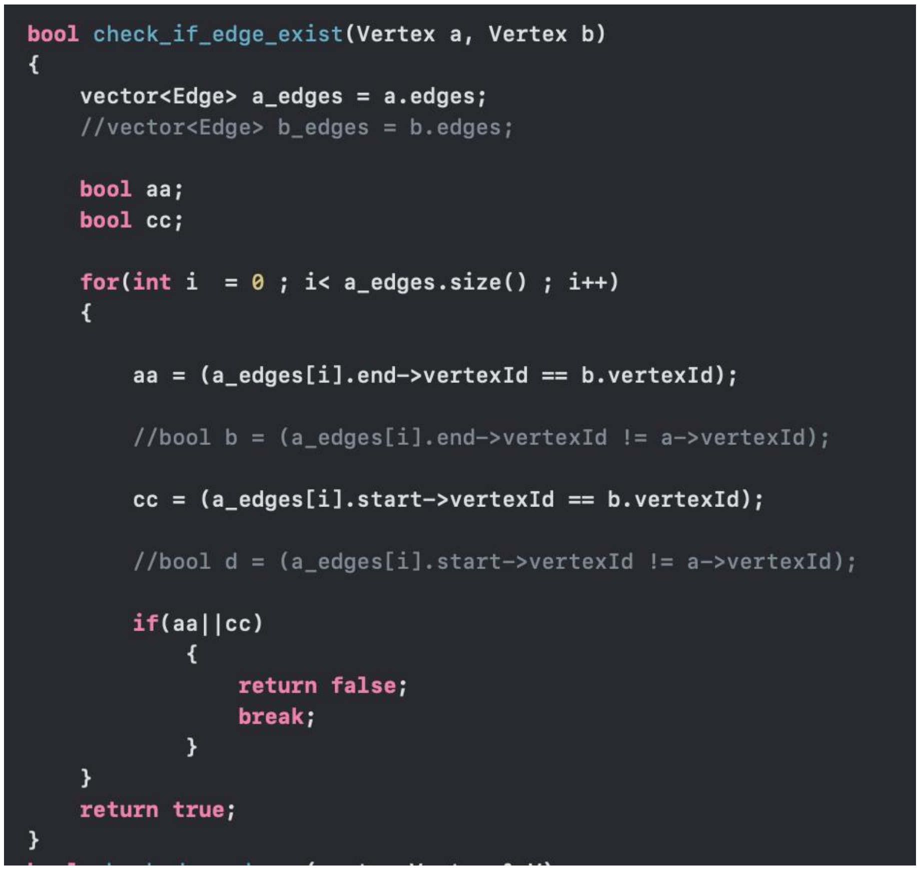
}

Else

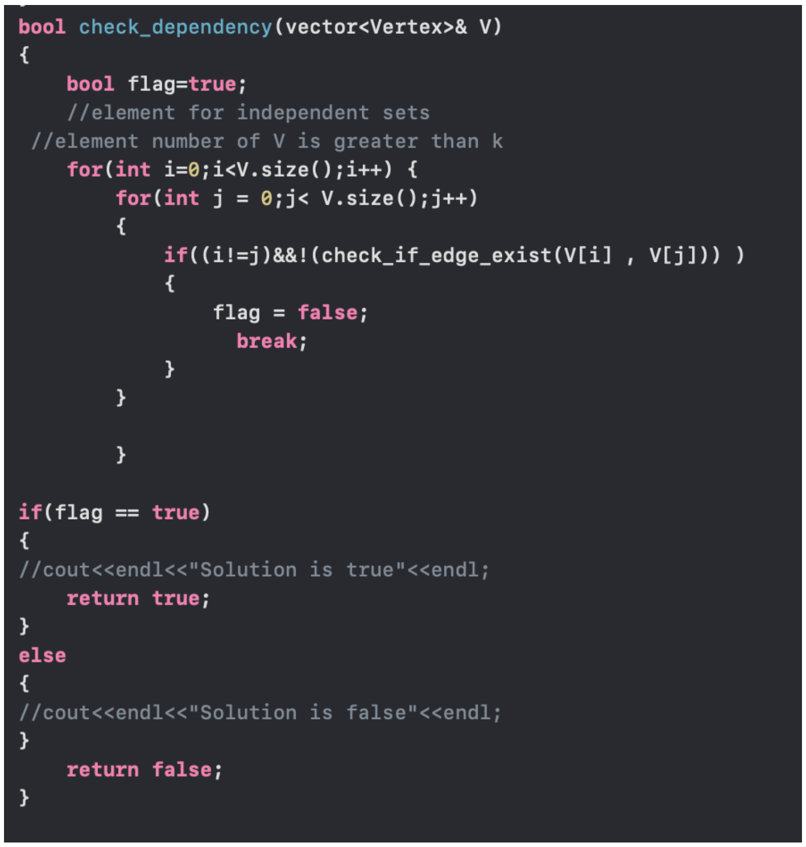
{

cout<<"Solution is false";

}



```
bool check_if_edge_exist(Vertex a, Vertex b)
{
    vector<Edge> a_edges = a.edges;
    //vector<Edge> b_edges = b.edges;

    bool aa;
    bool cc;

    for(int i  = 0 ; i< a_edges.size() ; i++)
    {

        aa = (a_edges[i].end->vertexId == b.vertexId);

        //bool b = (a_edges[i].end->vertexId != a->vertexId);

        cc = (a_edges[i].start->vertexId == b.vertexId);

        //bool d = (a_edges[i].start->vertexId != a->vertexId);

        if(aa||cc)
            {
                return false;
                break;
            }
    }
    return true;
}
```



```
bool check_dependency(vector<Vertex>& V)
{
    bool flag=true;
    //element for independent sets
 //element number of V is greater than k
    for(int i=0;i<V.size();i++) {
        for(int j = 0;j< V.size();j++)
        {
            if((i!=j)&&!(check_if_edge_exist(V[i] , V[j])) )
            {
                flag = false;
                  break;
            }
        }

        }

if(flag == true)
{
//cout<<endl<<"Solution is true"<<endl;
    return true;
}
else
{
//cout<<endl<<"Solution is false"<<endl;
}
    return false;
}
```

In conclusion, it can be said that independent set is in NP.

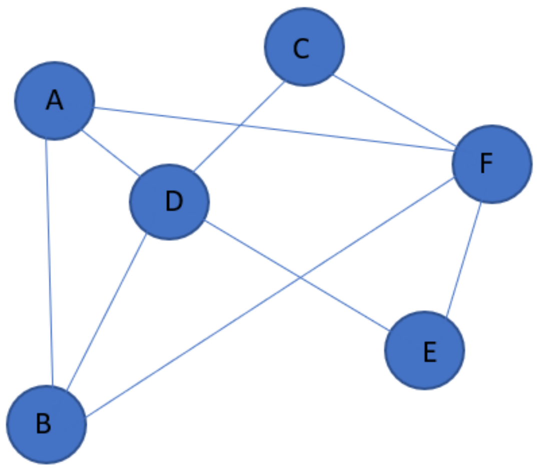
b) An NP complete problem can be reduced to Independent Set in polynomial time.

To be able to prove whether Independent Set is NP Hard, a reduction algorithm from Clique problem which is a known NP-Complete problem to the Independent Set problem is performed.

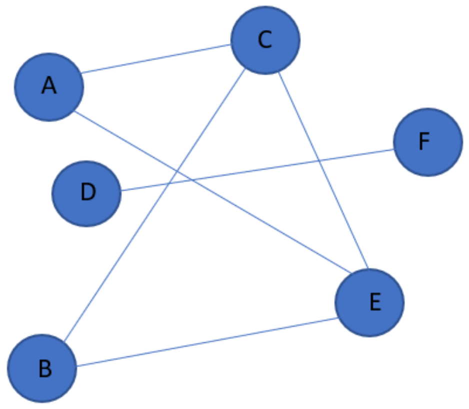
A clique is a subgraph of a graph in which all vertices are connected with each other.

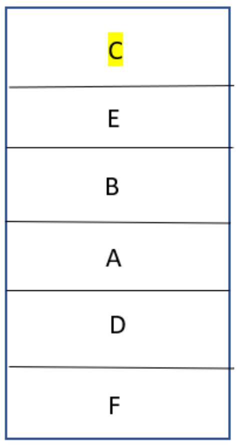
Steps:

1.Firstly, the reverse of a given graph G is taken(G') which would mean that every existing edge is broken and every non-existing edge is added.



complementary

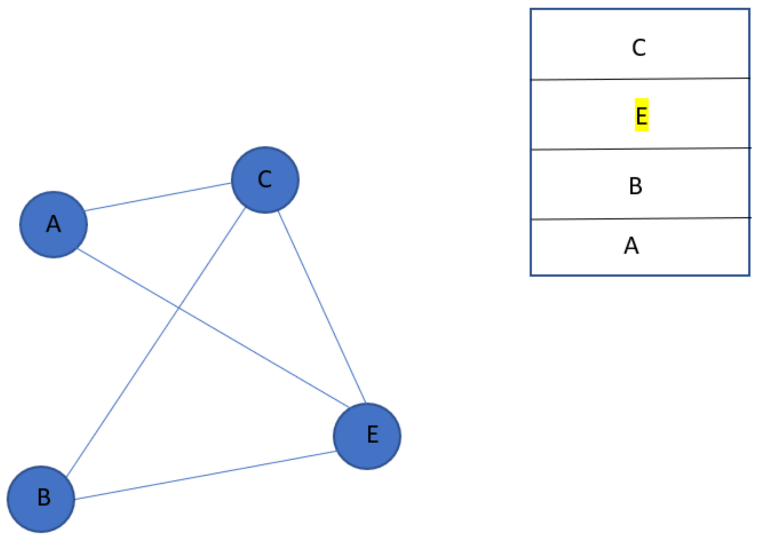




2.If we can find the clique with k number of vertices in our complementary graph G', where each vertex is connected with other vertices with edges. The presence of edges in our complementary graph G' show us that, the edges are non-existent in our graph G. Therefore, since there exists edges between the k adjacent vertices in our complementary graph G', this would mean these k vertices would give us an independent set of size k as a subset of our graph G.

3.Let us first prepare a priority list from the vertex having the least edged vertices to most edged vertices.

4.Finally, we can start selecting vertices from the list and remove the vertex if that vertex is selected and we should repeat this procedure until there is no vertex left on our priority list.



2)ALGORITHM DESCRIPTION

After doing intensive research, no algorithm was found to identify greatest independent set in a given vertex with polynomial time. However, suitable heuristic algorithm was found to solve this problem in polynomial time but it should also be considered that the solution is not correct every time. It can be said that this is a greedy algorithm in order to find the maximum independent set of a given graph.

The Greedy Algorithm Steps:

1. The algorithm creates a vertex array with given inputs and for these vertices it creates an edge array for each vertex. Therefore, our graph class would be completed.
2. Then it finds the vertex with most edges and mark it with 1.All the other neighbors will be marked as 0. It continues to find non marked vertex with most edges and mark their neighbors as 0. Until there is not any vertex unmarked.
3. Then it creates a set with marked 1's and it is the maximum independent set.

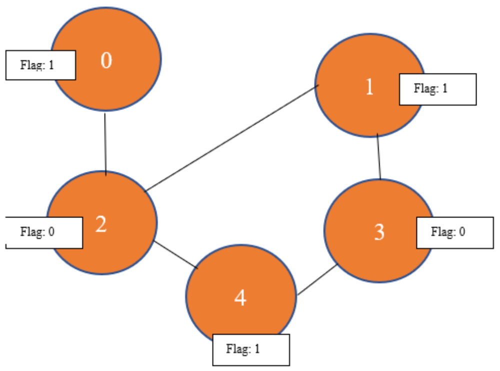
Further information about algorithm: Although this algorithm always finds an independent set it cannot assure that the set found is a maximum independent set since it is a greedy algorithm.

3)ALGORITHM ANALYSIS

This algorithm runs in O(VlogV + E), which is in polynomial time. Thus, IS ∈ NP.

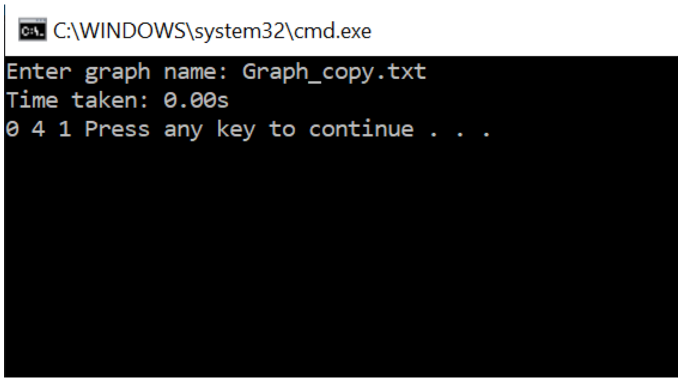
The working manner of the following code:

Suppose we are given a graph G as follows;



First it finds the vertex with most edges which is Vertex 1. Then marks it 1. Then marks it's neighbors as 0 which are Vertex 2 and Vertex 3. Then it finds another vertex with the most edges which is vertex 4 and marks it 1(check their neigbours). Finally it marks Vertex 0 with 1.Finally, a set with marked 1 vertices is created.

The output would be as follows;

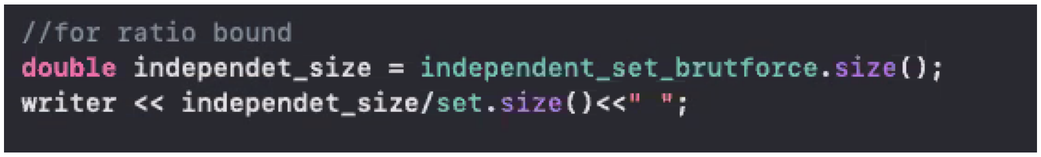


C:\WINDOWS\system32\cmd.exe

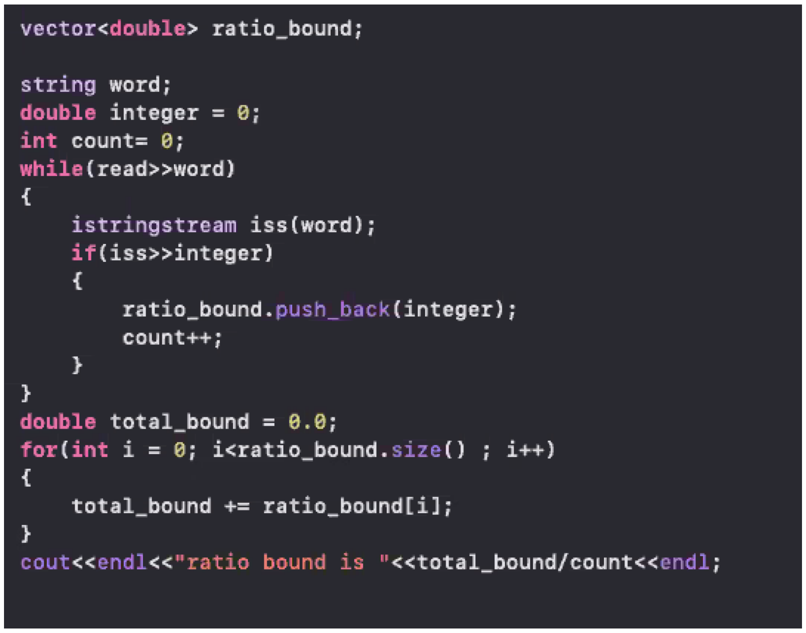
```
Enter graph name: Graph_copy.txt
Time taken: 0.00s
0 4 1 Press any key to continue . . .
```

4) EXPERIMENTAL ANALYSIS

a)RATIO BOUND:



```
//for ratio bound
double independet_size = independent_set_brutforce.size();
writer << independet_size/set.size()<<" ";
```

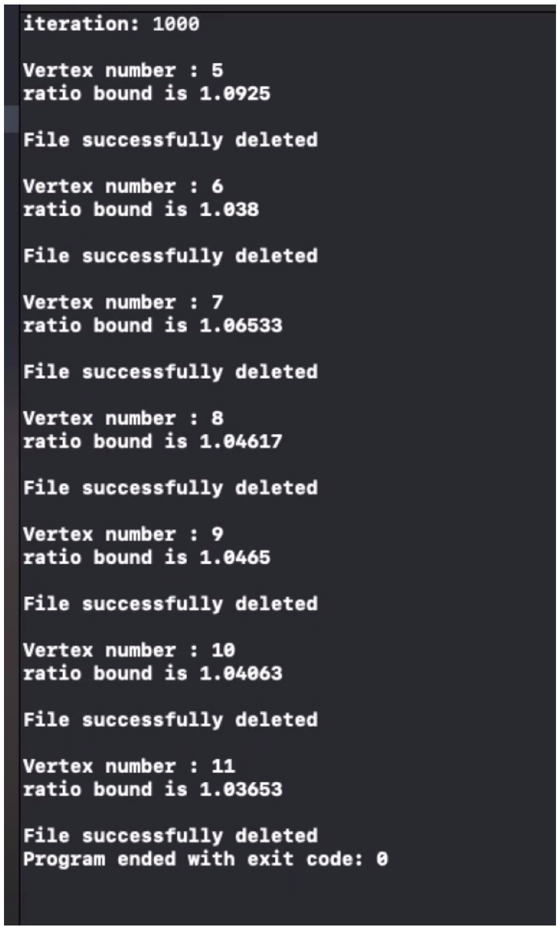


```
vector<double> ratio_bound;

string word;
double integer = 0;
int count= 0;
while(read>>word)
{
    istringstream iss(word);
    if(iss>>integer)
    {
        ratio_bound.push_back(integer);
        count++;
    }
}
double total_bound = 0.0;
for(int i = 0; i<ratio_bound.size() ; i++)
{
    total_bound += ratio_bound[i];
}
cout<<endl<<"ratio bound is "<<total_bound/count<<endl;
```

We say that an approximation algorithm for the problem has a ***ratio bound*** of $\rho(n)$ if for any input of size n, the cost C of the solution produced by the approximation algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) . \qquad (37.1)$$

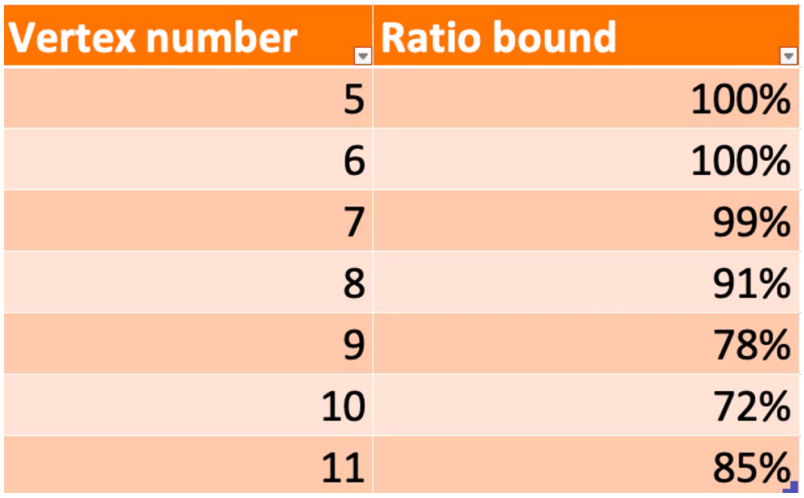
->The sizes of our optimal solution and approximation algorithms were compared, and the results can be seen below;



```
iteration: 1000

Vertex number : 5
ratio bound is 1.0925

File successfully deleted

Vertex number : 6
ratio bound is 1.038

File successfully deleted

Vertex number : 7
ratio bound is 1.06533

File successfully deleted

Vertex number : 8
ratio bound is 1.04617

File successfully deleted

Vertex number : 9
ratio bound is 1.0465

File successfully deleted

Vertex number : 10
ratio bound is 1.04063

File successfully deleted

Vertex number : 11
ratio bound is 1.03653

File successfully deleted
Program ended with exit code: 0
```

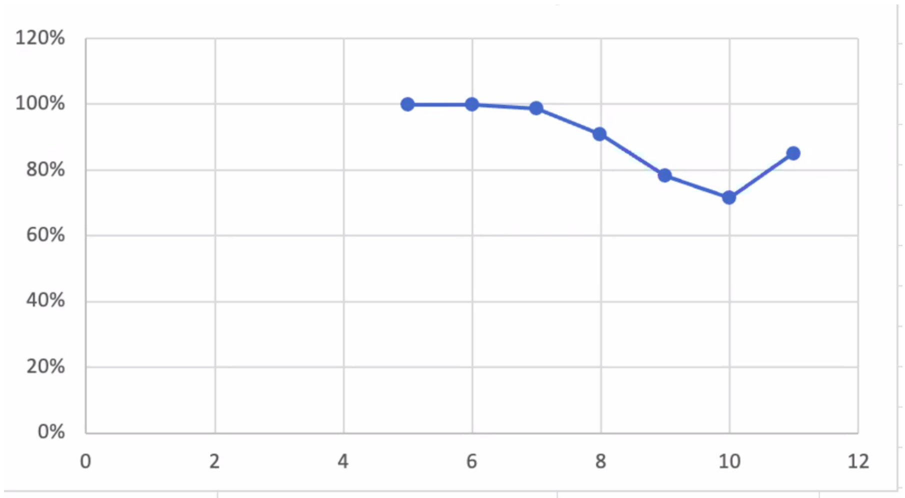
Approximation Algorithm finds near optimal solutions,used for finding the exact solutions to problems that require exponential amount of time, but doesn’t guarantee finding a correct solution.

Brute Force Algorithm ; Brute force algorithms are simple to implement but computationally intensive to run. They can be considered as reasonable solutions when n is small but for moderately large values of n the solutions become too intensive to produce results in a reasonable time frame.As a conclusion the optimal solution has more correctness.



Vertex number	Ratio bound
5	100%
6	100%
7	99%
8	91%
9	78%
10	72%
11	85%

Iteration number and edge number have not been changed which the value for iteration number is 20 and edge number is 150.

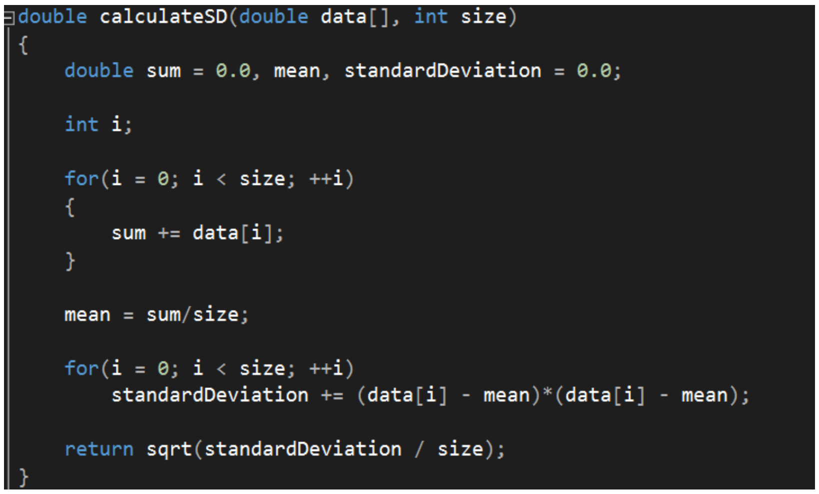


When we increase vertex number of the graph, our heuristic algorithm's correctness is decreasing and when we increase to a higher vertex count correctness of algorithm will be 0.

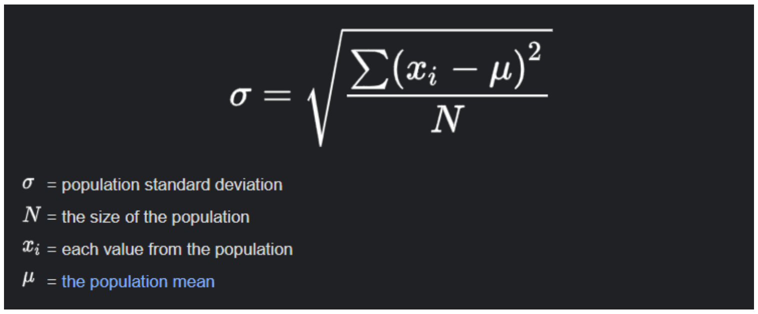
b) Running Time Experimental Analysis

In order find running time complexity, data analysis tools were used such as standard deviation, standard error, sample mean and confidence level intervals(%90 and %95 each upper and lower bounds).

1)Standard Deviation



```
double calculateSD(double data[], int size)
{
    double sum = 0.0, mean, standardDeviation = 0.0;

    int i;

    for(i = 0; i < size; ++i)
    {
        sum += data[i];
    }

    mean = sum/size;

    for(i = 0; i < size; ++i)
        standardDeviation += (data[i] - mean)*(data[i] - mean);

    return sqrt(standardDeviation / size);
}
```



$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

σ = population standard deviation

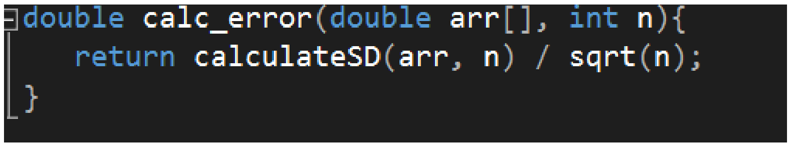
N = the size of the population

x_i = each value from the population

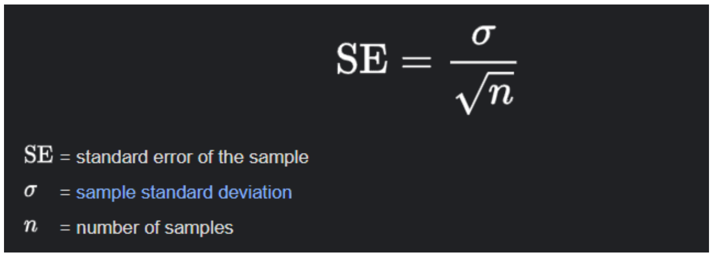
μ = the population mean

Standard deviation has been calculated with the code mentioned above.

2)Standard Error



```
double calc_error(double arr[], int n){
    return calculateSD(arr, n) / sqrt(n);
}
```



$$\mathrm{SE} = \frac{\sigma}{\sqrt{n}}$$

SE = standard error of the sample

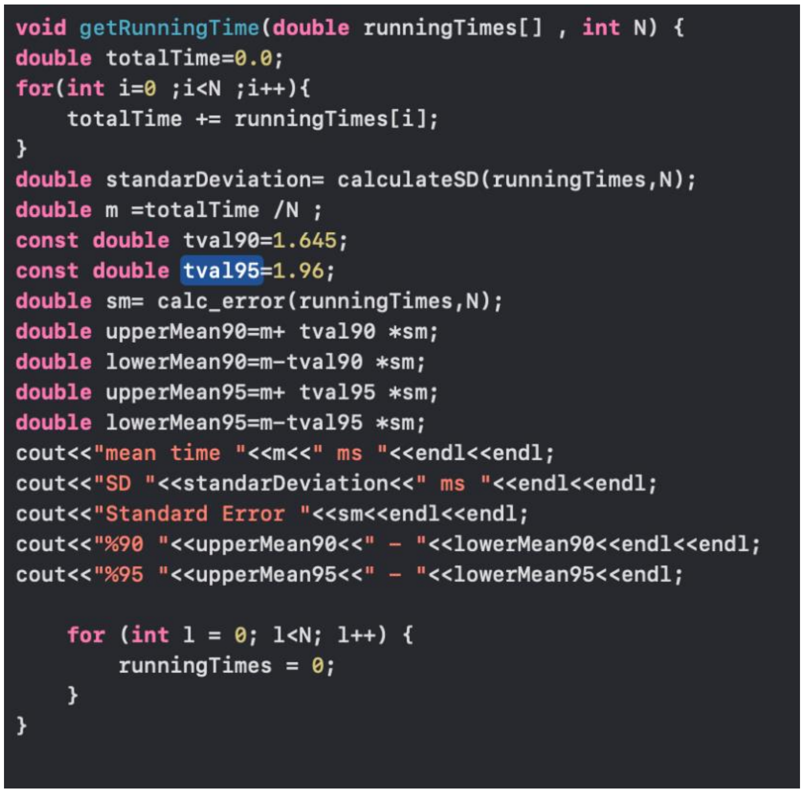
σ = sample standard deviation

n = number of samples

The standard error has been calculated with the code mentioned above.

3)Running Time Algorithm

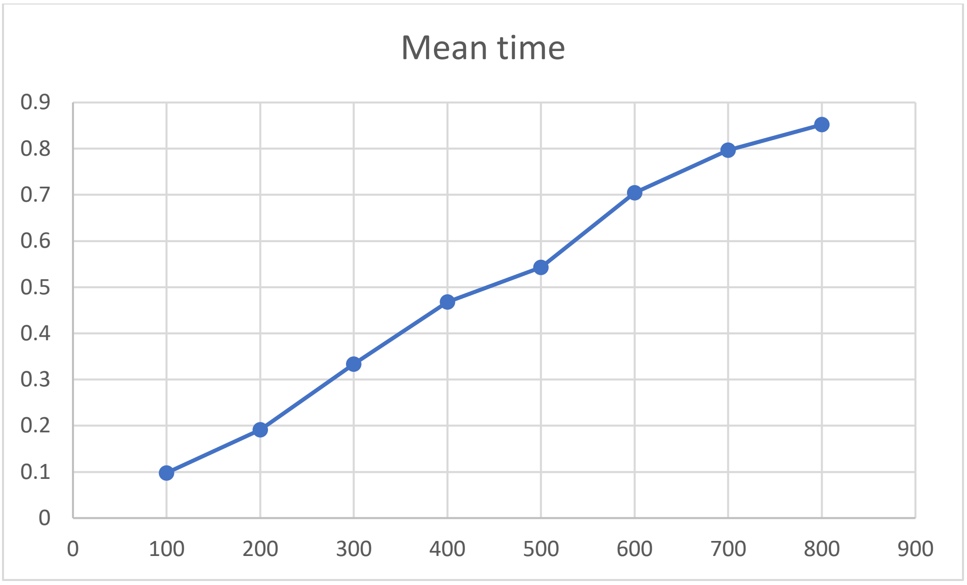
In the algorithm below, to be able to make it simpler t-value for %90 confidence interval has been taken as 1.645 and t-value for %95 confidence interval has been taken as 1.96.The algorithm below displays the mean time of the total time in milliseconds, the standard deviation of the running time, the standard error of the running time, the confidence intervals for %90 and %95.



```
void getRunningTime(double runningTimes[] , int N) {
double totalTime=0.0;
for(int i=0 ;i<N ;i++){
    totalTime += runningTimes[i];
}
double standarDeviation= calculateSD(runningTimes,N);
double m =totalTime /N ;
const double tval90=1.645;
const double tval95=1.96;
double sm= calc_error(runningTimes,N);
double upperMean90=m+ tval90 *sm;
double lowerMean90=m-tval90 *sm;
double upperMean95=m+ tval95 *sm;
double lowerMean95=m-tval95 *sm;
cout<<"mean time "<<m<<" ms "<<endl<<endl;
cout<<"SD "<<standarDeviation<<" ms "<<endl<<endl;
cout<<"Standard Error "<<sm<<endl<<endl;
cout<<"%90 "<<upperMean90<<" - "<<lowerMean90<<endl<<endl;
cout<<"%95 "<<upperMean95<<" - "<<lowerMean95<<endl;

    for (int l = 0; l<N; l++) {
        runningTimes = 0;
    }
}
```

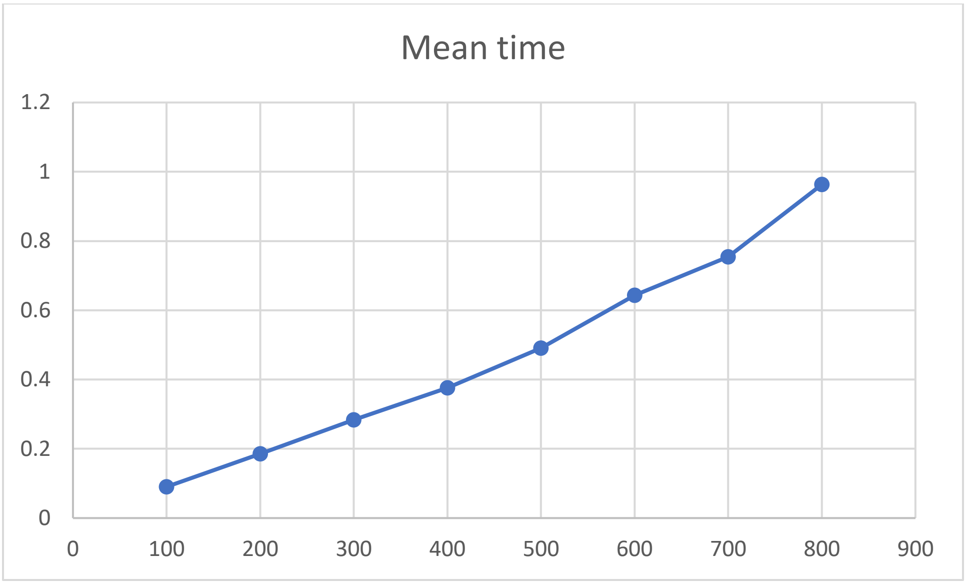
Edges Constant (E = 400) with iteration number 100

Vertex Number	Mean Time	Standard Deviation	Standard Error	%90 Confidence Level	%95 Confidence Level
100	0.09787	0.0268334	0.00268334	0.102284 - 0.0934559	0.103129 - 0.0926106
200	0.19088	0.0404614	0.00404614	0.197536 - 0.184224	0.19881 - 0.18295
300	0.3332	0.0968152	0.00968152	0.349126 - 0.317274	0.352176 - 0.314224
400	0.46785	0.109066	0.0109066	0.485791 - 0.449909	0.489227 - 0.446473
500	0.54297	0.120798	0.0120798	0.562841 - 0.523099	0.566646 - 0.519294
600	0.70419	0.162961	0.0162961	0.730997 - 0.677383	0.73613 - 0.67225
700	0.79637	0.1996	0.01996	0.829204 - 0.763536	0.835492 - 0.757248
800	0.85189	0.186582	0.0186582	0.882583 - 0.821197	0.88846 - 0.81532



Edges Constant (E = 400) with iteration number 5000

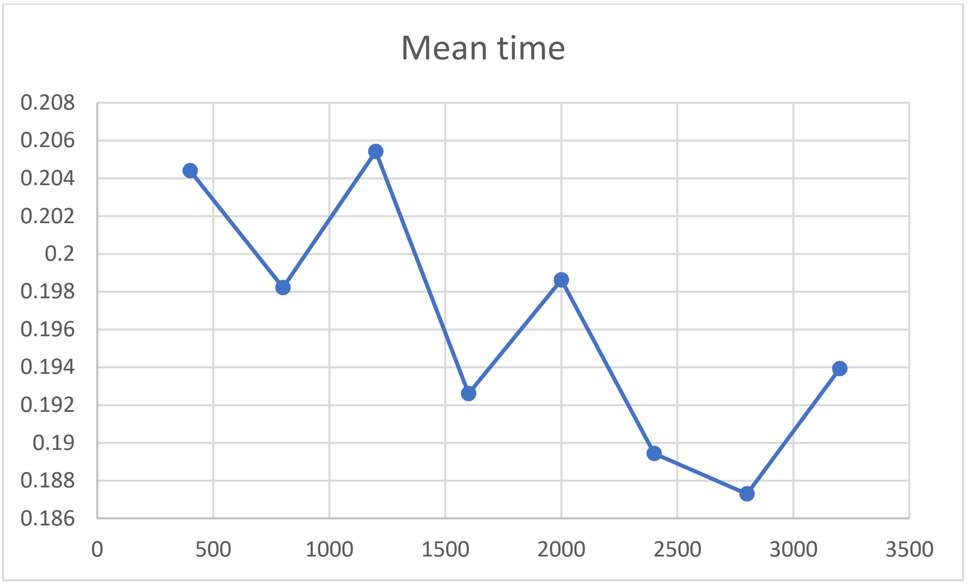
Vertex Number	Mean Time	Standard Deviation	Standard Error	%90 Confidence Level	%95 Confidence Level
100	0.0903154	0.0256103	0.000362184	0.0909112 - 0.0897196	0.0910253 - 0.0896055
200	0.186041	0.0532237	0.000752697	0.187279 - 0.184803	0.187516 - 0.184566
300	0.283575	0.0735259	0.00103981	0.285286 - 0.281865	0.285613 - 0.281537
400	0.375974	0.0950455	0.00134415	0.378186 - 0.373763	0.378609 - 0.37334
500	0.490553	0.123379	0.00174485	0.493424 - 0.487683	0.493973 - 0.487133
600	0.643107	0.173897	0.00245928	0.647153 - 0.639062	0.647928 - 0.638287
700	0.754773	0.194599	0.00275205	0.7593 - 0.750246	0.760167 - 0.749379
800	0.963011	0.250611	0.00354417	0.968842 - 0.957181	0.969958 - 0.956065



The graph illustrates the increase in vertex number effect on time complexity. Line's slope is similar to linear, but it is faster growing. We can observe V*log(V) with this graph which can validate our algorithm's complexity.

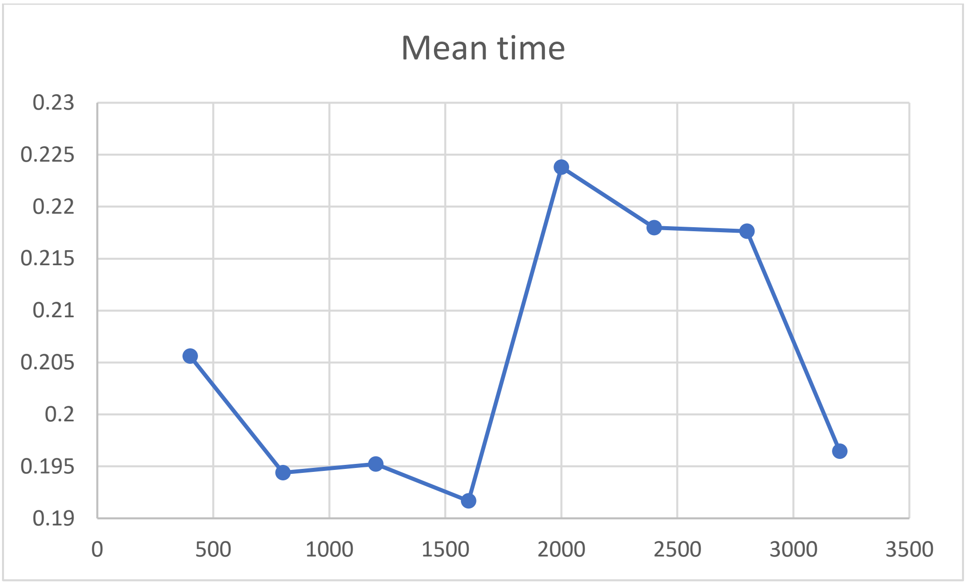
Vertices Constant (V = 200) iteration number 100.

Edge Number	Mean Time	Standard Deviation	Standard Error	%90 Confidence Level	%95 Confidence Level
400	0.20441	0.0702702	0.00702702	0.215969 - 0.192851	0.218183 - 0.190637
800	0.19822	0.0482383	0.00482383	0.206155 - 0.190285	0.207675 - 0.188765
1200	0.20542	0.0736103	0.00736103	0.217529 - 0.193311	0.219848 - 0.190992
1600	0.19261	0.0513164	0.00513164	0.201052 - 0.184168	0.202668 - 0.182552
2000	0.19862	0.0557964	0.00557964	0.207799 - 0.189441	0.209556 - 0.187684
2400	0.18944	0.0467712	0.00467712	0.197134 - 0.181746	0.198607 - 0.180273
2800	0.18729	0.0383993	0.00383993	0.193607 - 0.180973	0.194816 - 0.179764
3200	0.19393	0.0626574	0.00626574	0.204237 - 0.183623	0.206211 - 0.181649



Vertices Constant (V = 200) iteration number 5000.

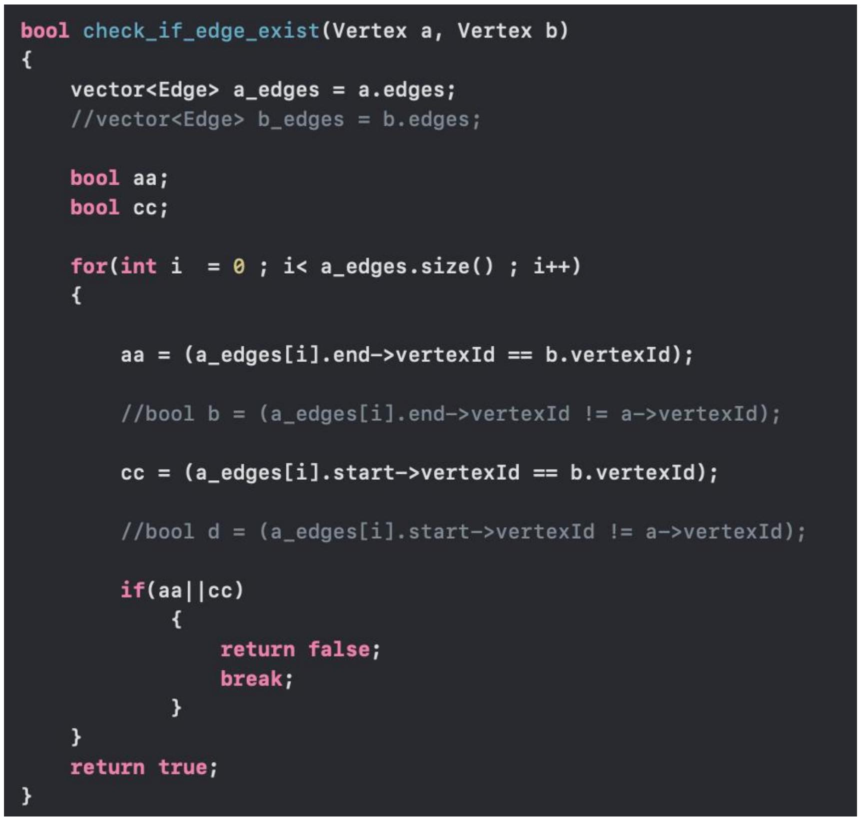
Edge Number	Mean Time	Standard Deviation	Standard Error	%90 Confidence Level	%95 Confidence Level
400	0.205623	0.0669703	0.000947103	0.207181 - 0.204065	0.20748 - 0.203767
800	0.194388	0.0590528	0.000835132	0.195762 - 0.193014	0.196025 - 0.192751
1200	0.19524	0.0630259	0.000891321	0.196706 - 0.193774	0.196987 - 0.193493
1600	0.191674	0.0696462	0.000984946	0.193294 - 0.190054	0.193604 - 0.189743
2000	0.223796	0.0775779	0.00109712	0.225601 - 0.221991	0.225947 - 0.221646
2400	0.217965	0.0714496	0.00101045	0.219627 - 0.216302	0.219945 - 0.215984
2800	0.217642	0.0737929	0.00104359	0.219359 - 0.215926	0.219688 - 0.215597
3200	0.196465	0.0634916	0.000897907	0.197942 - 0.194988	0.198225 - 0.194705



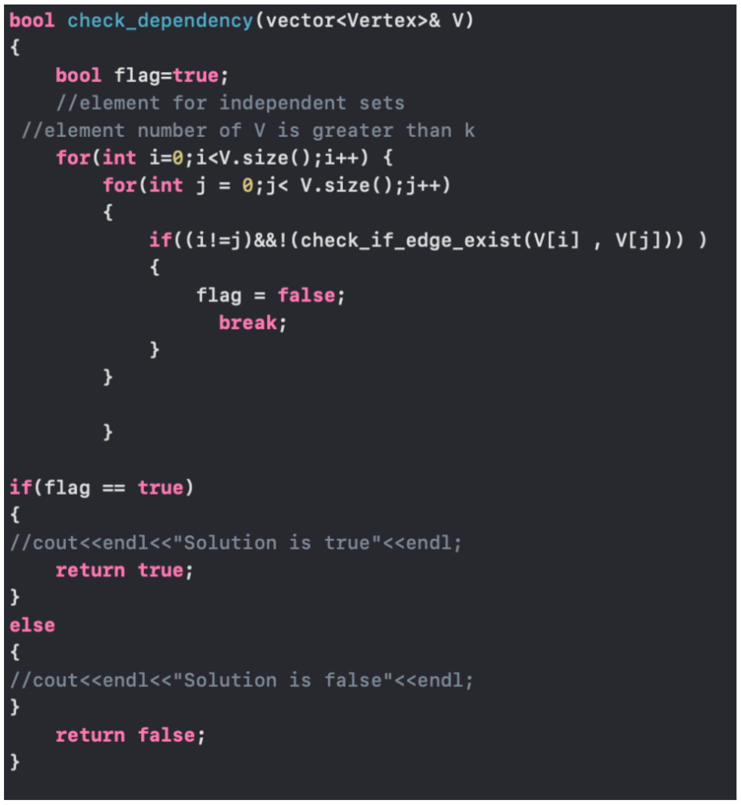
This graph shows us that our mean time have a global maximum near 2000 edge. In a first look it may be confusing but when investigated the reason behind that becomes clearer. As mentioned above, our algorithm first finds the vertex with the most number of edges and marks it 1 then marks its neighbors with 0 then continues until there is no vertex left unmarked. If we have huge amount of edges, there will be more neighborhood. Our algorithm will mark nearly all of them at the first iteration and there will be very little amount of unmarked vertex. If we have very little number of edges, then we will not be trying finding neighbors and we just mark them 1. Time complexity spending for finding neighbors will decrease. That's why our graph has a global maximum at point 2000.

c)Correctness

The heuristic algorithm for finding maximum independent set has a polynomial time complexity but in order to succeed this polynomial time complexity, our algorithm is decreasing correctness percentage. This algorithm does not give the maximum independent set from given graph with %100 correctness.

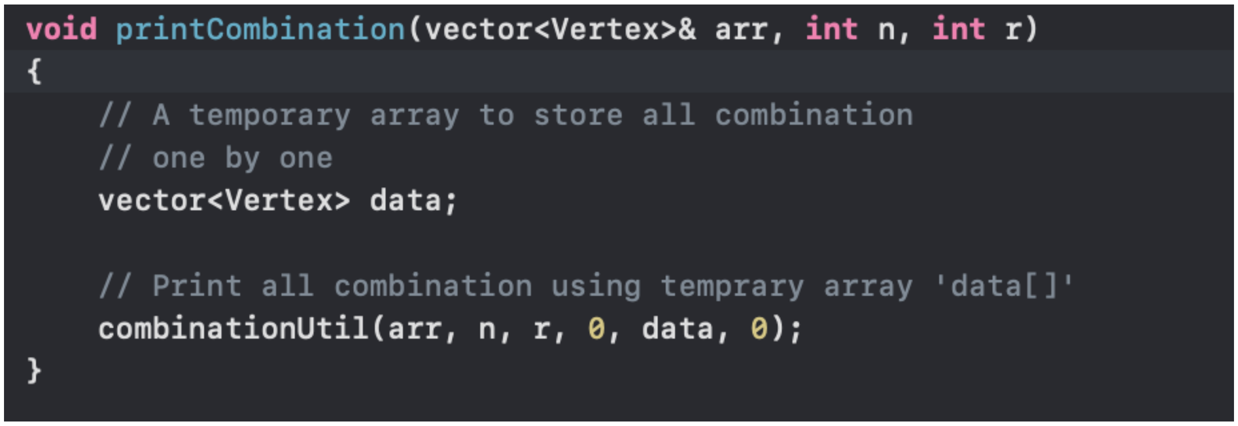


```
bool check_if_edge_exist(Vertex a, Vertex b)
{
    vector<Edge> a_edges = a.edges;
    //vector<Edge> b_edges = b.edges;

    bool aa;
    bool cc;

    for(int i  = 0 ; i< a_edges.size() ; i++)
    {

        aa = (a_edges[i].end->vertexId == b.vertexId);

        //bool b = (a_edges[i].end->vertexId != a->vertexId);

        cc = (a_edges[i].start->vertexId == b.vertexId);

        //bool d = (a_edges[i].start->vertexId != a->vertexId);

        if(aa||cc)
            {
                return false;
                break;
            }
    }
    return true;
}
```

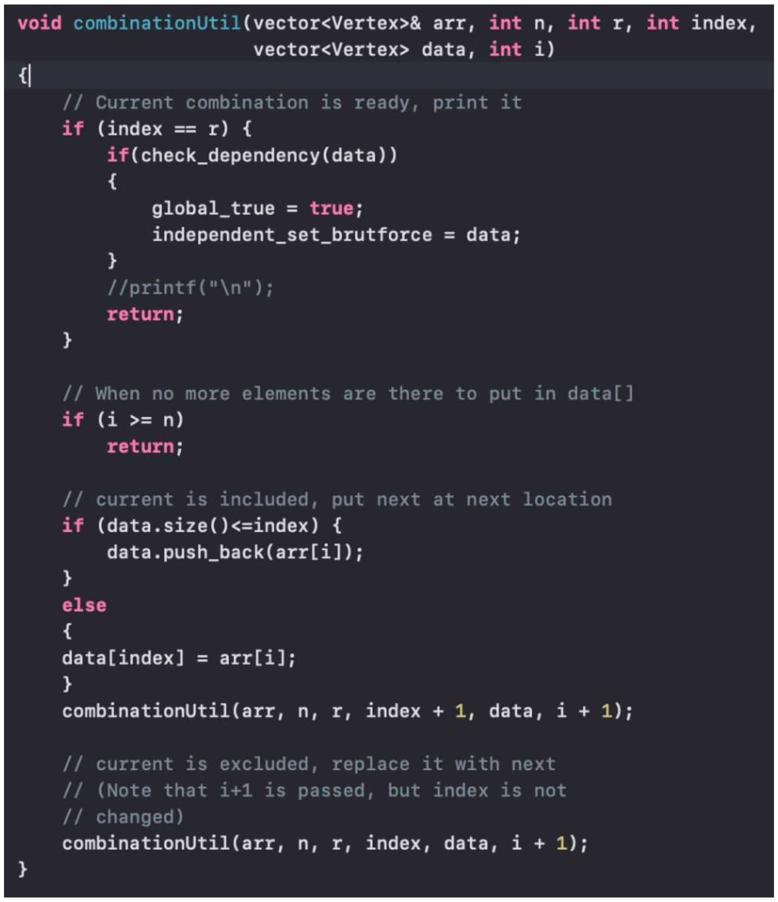


```
bool check_dependency(vector<Vertex>& V)
{
    bool flag=true;
    //element for independent sets
 //element number of V is greater than k
    for(int i=0;i<V.size();i++) {
        for(int j = 0;j< V.size();j++)
        {
            if((i!=j)&&!(check_if_edge_exist(V[i] , V[j])) )
            {
                flag = false;
                  break;
            }
        }

        }

if(flag == true)
{
//cout<<endl<<"Solution is true"<<endl;
    return true;
}
else
{
//cout<<endl<<"Solution is false"<<endl;
}
    return false;
}
```

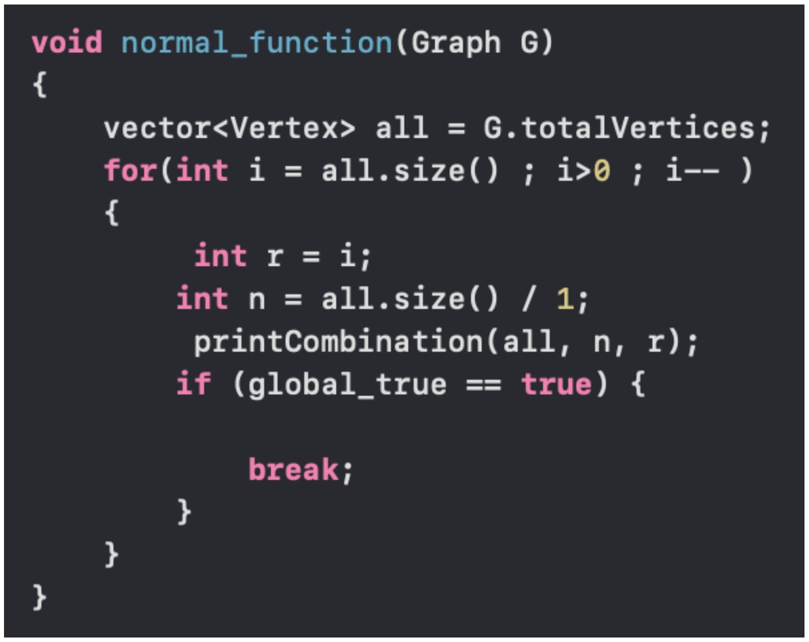
The algorithm above is checking whether a given set is an independent set or not. As mentioned above, this algorithm checks if chosen vertices have edges between them. If there is no edge between any vertex then our set is independent set. However, with this algorithm we did not check whether it is the maximum independent set or not. In order to find maximum independent set with %100 certainty we have find all subsets of given vertex indexes. For example, {1,2,3} => {1,2,3} , {1,2} , {2,3}, {1,3} , {1} , {2} , {3} , {}. Then we have to check each of the subset starting from largest, if it is an independent set when we find independent set it will be maximum independent set with %100 certainty.



```
void printCombination(vector<Vertex>& arr, int n, int r)
{
    // A temporary array to store all combination
    // one by one
    vector<Vertex> data;

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, n, r, 0, data, 0);
}
```



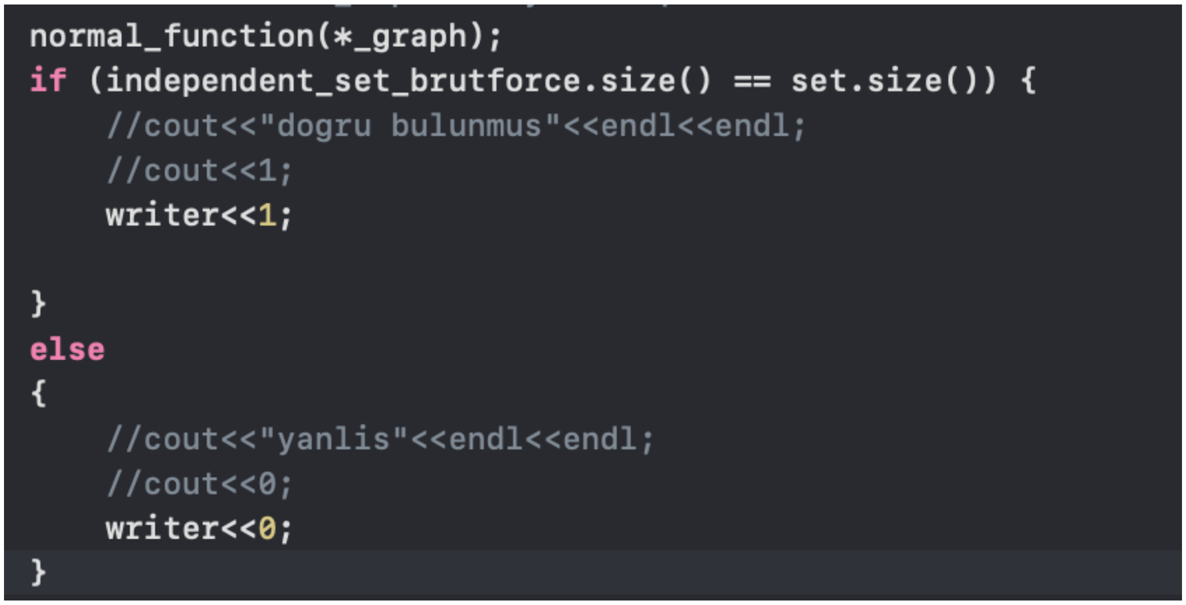
```
void combinationUtil(vector<Vertex>& arr, int n, int r, int index,
                     vector<Vertex> data, int i)
{
    // Current combination is ready, print it
    if (index == r) {
        if(check_dependency(data))
        {
            global_true = true;
            independent_set_brutforce = data;
        }
        //printf("\n");
        return;
    }

    // When no more elements are there to put in data[]
    if (i >= n)
        return;

    // current is included, put next at next location
    if (data.size()<=index) {
        data.push_back(arr[i]);
    }
    else
    {
    data[index] = arr[i];
    }
    combinationUtil(arr, n, r, index + 1, data, i + 1);

    // current is excluded, replace it with next
    // (Note that i+1 is passed, but index is not
    // changed)
    combinationUtil(arr, n, r, index, data, i + 1);
}
```



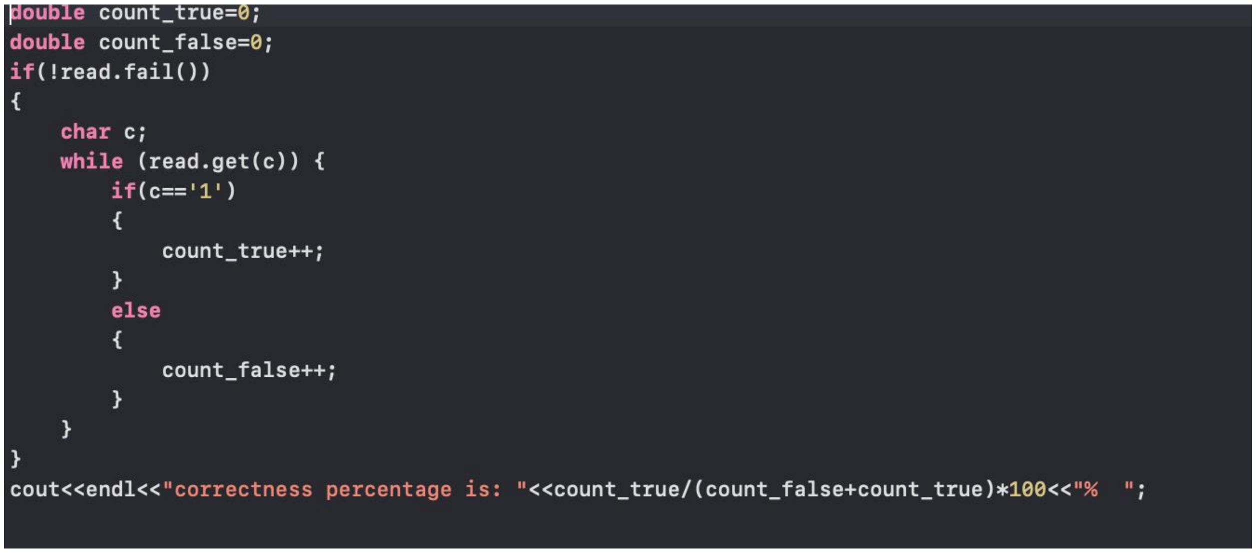
```
void normal_function(Graph G)
{
    vector<Vertex> all = G.totalVertices;
    for(int i = all.size() ; i>0 ; i-- )
    {
         int r = i;
        int n = all.size() / 1;
         printCombination(all, n, r);
        if (global_true == true) {

            break;
        }
    }
}
```

This algorithm finds all subsets of a given graph starting with largest one. This algorithm has a complexity 2 to the power of vertex size of given graph.

Testing:

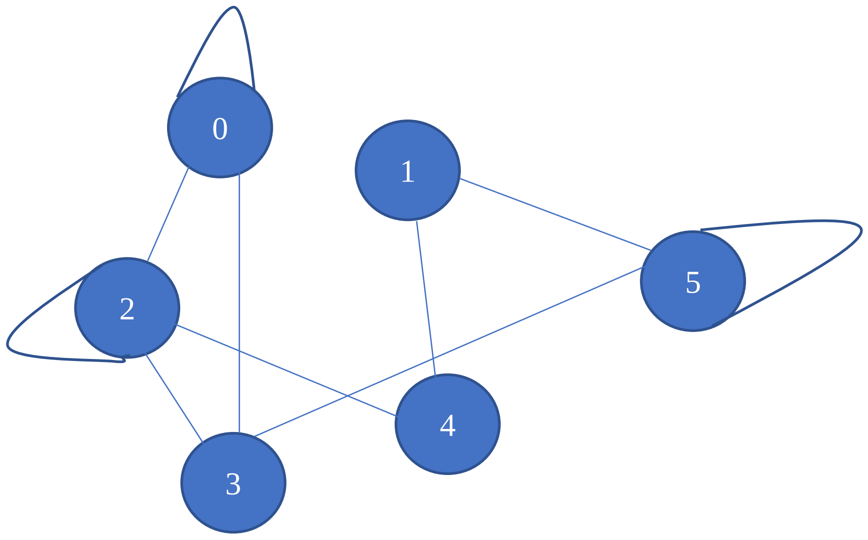


```
normal_function(*_graph);
if (independent_set_brutforce.size() == set.size()) {
    //cout<<"dogru bulunmus"<<endl<<endl;
    //cout<<1;
    writer<<1;

}
else
{
    //cout<<"yanlis"<<endl<<endl;
    //cout<<0;
    writer<<0;
}
```

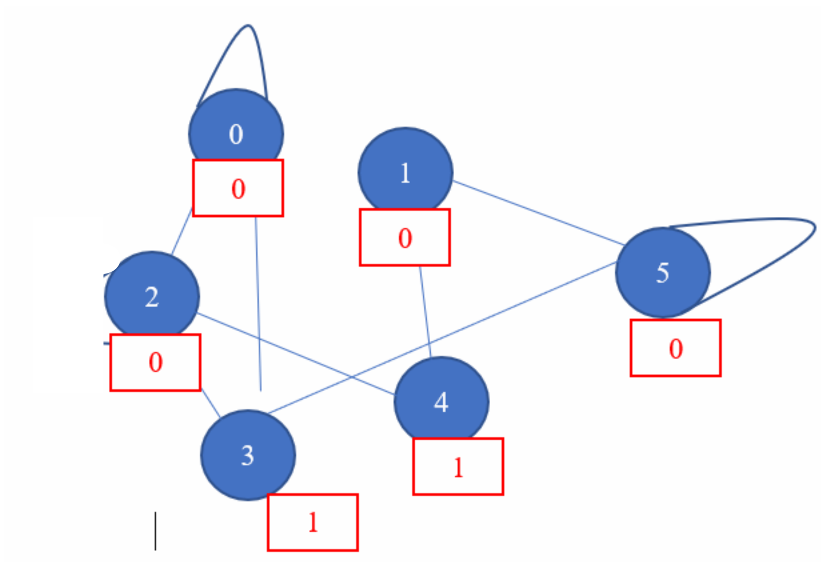


```
double count_true=0;
double count_false=0;
if(!read.fail())
{
    char c;
    while (read.get(c)) {
        if(c=='1')
        {
            count_true++;
        }
        else
        {
            count_false++;
        }
    }
}
cout<<endl<<"correctness percentage is: "<<count_true/(count_false+count_true)*100<<"%  ";
```

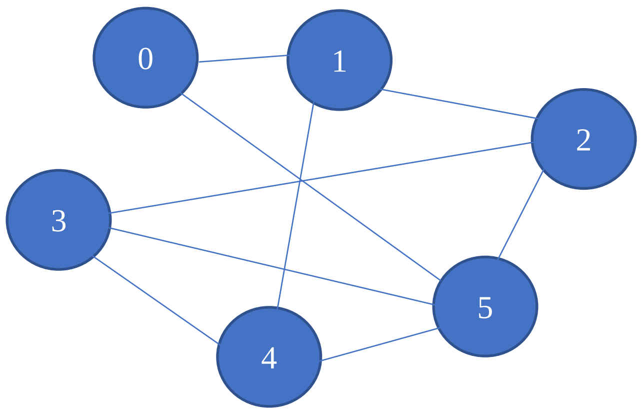
This algorithm compares the size of maximum independent set found by heuristic algorithm and brute force algorithm which we describe detailly in correctness part.



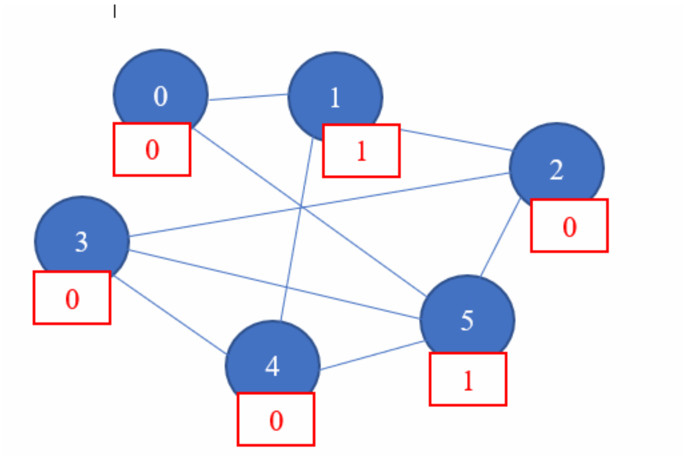
FAILED: Our algorithm suggests that 4 and 3 form an independent set. However the optimal solution for this would be 0,4 and 5.



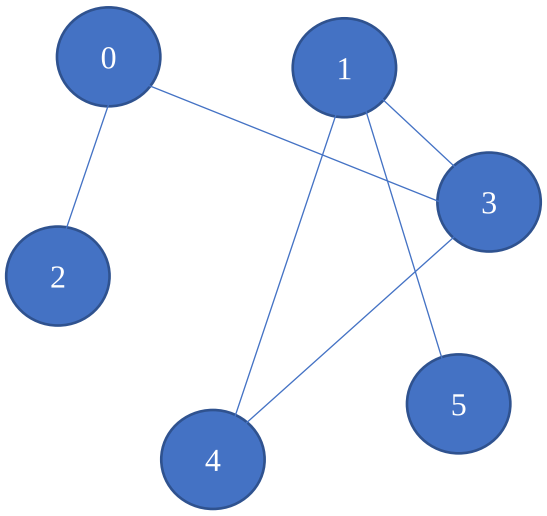
Our algorithm suggested the graph above with their marked values. Therefore, this would give us the set 4 and 3 however it is clearly seen that the maximum independent set is 0,4 and 5.



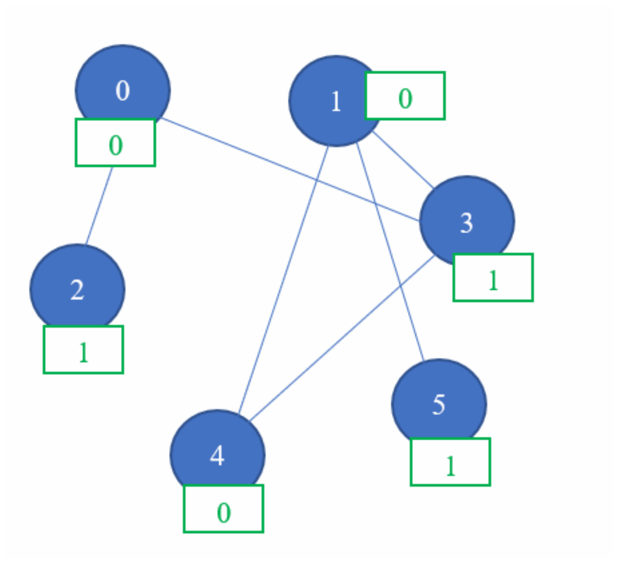
FAILED: Our algorithm suggests that 1 and 5 form an independent set. However, the optimal solution for this would be 0,4 and 2.



Our algorithm suggested the graph above with their marked values. Therefore, this would give us the set 1 and 5 however it is clearly seen that the maximum independent set is 0,4 and 2.



SUCCEED: Our algorithm does work correctly in this case and therefore the correct maximum independent set is found which is 3,5,2.



DISCUSSION:

In conclusion, we proved that Maximum Independent Set is NP-Complete problem which is reduced from Clique. An NP complete problem can be reduced to Independent Set in polynomial time by sacrificing correctness percentage. There are heuristic algorithms that are fast and it doesn’t guarantee to find an optimum solution and it is used for these NP problems and trades optimality/correctness for speed. We found an heuristic algorithm which solved it in polynomial time but the solution was not correct every time. This is a greedy algorithm for finding maximum independent set. Further information about algorithm: This algorithm always found an independent set but it did not guarantee to find a maximum independent set because it is a greedy algorithm.This algorithm runs in $O(V\log V + E)$, which is in polynomial time. Thus, IS ∈ NP.

In test phase ,we kepth either vertex or edge number constant with changing the other one. Moreover at highest and lowest edge numbers , our algorithm was faster because if we had huge amount of edges, there would be more neighborhood. Our algorithm would mark nearly all of them at the first iteration and in the end there would be very little amount of unmarked vertex. If we had very little number of edges, then we would not try to find neighbors and we would just mark them with 1.

In order to test heuristic algorithm’s correctness a brute force algorithm (optimal solution) was also added.

REFERENCES:

https://www.cs.utexas.edu/~mitra/csSpring2017/cs313/lectures/algo_classes.html

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap37.html

https://github.com/eshirima/Maximum-Independent-Set