CS301

2022 - 2023 Spring

Project Report

Group 122

Zeynep Türkmen

Kaan Adalılar

1- Problem Description

Formal Definition:

Input: An n-node undirected graph $G(V,E)$ with node set V and edge set E; a positive integer k with $k<=n$.

Question: Does G contain a k-clique, i.e. a subset W of the nodes V such that W has size k and for each distinct pair of nodes u, v in W, $\{u,v\}$ is an edge of G?

Intuitive Definition:

Clique is a problem in graph theory that checks whether there is a group of vertices in an undirected graph in which they are all connected to one another. In other words, it asks if there is a subgraph of size k (where k is an input) that all the vertices are connected to every other vertex in that subgraph. So basically it aims to check whether there is a clique of a given size and if there is, to find it.

Applications:

In real life, this clique problem can be used to analyze social networks of people where the vertices of the graph are the people and the edges are their relationships. With the help of this problem an inner circle of relationships may be determined where each person knows one another. So given a size k we may find an inner circle of k people that all know each other.

Similarly, it may be used in computational biology in order to determine the genes that interact with one another. In this case the vertices are the genes and the edges are connections. Again given a size k, we can find k genes that are connected. Considering the optimization version of this clique problem where k is the max possible, we may find the most amount of genes that are connected.

Hardness of the Clique Problem:

The decision version of the Clique problem was proven to be NP-Complete using the reduction technique. In 1972, Karp showed that the clique problem can be reduced to the SAT problem (Boolean Satisfiability) by creating a similar Boolean formula that is SATisfiable only when that graph has a clique of given size. He showed that in the following theorem:

3. CLIQUE
INPUT: graph G, positive integer k
PROPERTY: G has a set of k mutually adjacent nodes.

SATISFIABILITY $\propto$ CLIQUE

$N = \{\langle \sigma, i \rangle \mid \sigma \text{ is a literal and occurs in } C_i\}$

$A = \{\{\langle \sigma, i \rangle, \langle \delta, j \rangle\} \mid i \neq j \text{ and } \sigma \neq \bar{\delta}\}$

$k = p$, the number of clauses.

The proof can also be found in the following document:
(Karp, R.M. (1972). Reducibility among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds) Complexity of Computer Computations. The IBM Research Symposia Series. Springer, Boston, MA. https://doi.org/10.1007/978-1-4684-2001-2_9)

2- Algorithm Description

a) Brute Force Algorithm:

For the brute force algorithm, we found an implementation from GeeksForGeeks which can be accessed with this link:
https://www.geeksforgeeks.org/find-all-cliques-of-size-k-in-an-undirected-graph/

Here is the pseudo code version of the algorithm:

```
int store[MAX]; //stores the picked vertices

bool is_clique(int b) //size of current candidate clique

{

    // for all set of edges within the current selection
    for (int i = 1; i < b; i++)
       for (int j = i + 1; j < b; j++)
           if (any edge between j and i is missing)
               return false;
    }
    return true;
}

// Function to find all the cliques of size s
// i is the starting vertice to consider
// l is the amount of vertices already inserted into the store array
bool findCliques(int i, int l, int s)
{
      // Check if any vertices from i+1 can be inserted
      while(from starting vertex i, consider all vertices that weren't
      checked before and may form a clique of size s) //Not checking
      all subsets, only the ones of size s

       if (the vertex has sufficient degree){
            store[l] = j; // Insert the candidate vertex
            if (is_clique(l + 1)) // check if it forms a clique with
            the previous items
                 if (it's still not the desired size)
                     // Recursion to add more vertices
                     findCliques(j, l + 1, s);
                 else
                     Return true or the clique array itself (store[]).
        }
}
```

Explanation of the algorithm:

The brute force algorithm achieves to solve the problem by recursion and nested loops. It basically iterates over all possible subsets of vertices of size k (the requested size) in the graph and checks if all the possible edges between them exist or not. The is_clique function iterates over all sets of edges and checks if there is an edge between two vertices for all possible combinations of vertices, if any misconnection is determined, then the execution stops and the function returns false directly. The while loop in the findCliques function is being executed until there are no remaining unchecked vertices that may form a clique. Inside of the while loop there are nested if statements. Outer if statement checks whether the current vertex has a sufficient degree and if the vertex is appropriate for being an element for clique, it is inserted into the *store* set. The inner if statement checks whether the current vertex forms a clique with the previously added vertices or not. If this condition is satisfied, then the size of the clique set is checked. If the size is the same as the desired size, then the algorithm returns the clique array which means that the graph contains a k sized clique. If still the size is not satisfied, recursion gets involved and findCliques function calls itself to add more vertices. It continues to iterate over these steps until the clique set reaches the given size k.

When the mechanism is considered, it can be inferred that the algorithm is designed as a divide-and-conquer algorithm. The algorithm adds a vertex into the array and turns the problem into finding a clique of size k-1 using the recursive calls, thereby reducing the size of the problem. Turning it into smaller subproblems reduces the amount of computations because, instead of checking each subgraph of size k, we can stop iterating that specific subset if some items do not match. This helps with the efficiency of the algorithm.

Another important point is that this is an exponential algorithm. It is a brute force algorithm and in the worst case the algorithm needs to check each subgraph of size k, in order to make sure a clique exists or not. If the graph is of size n and the clique is of size k, in the worst case happens when k is around n/2 as the combination of n with k gives the greatest around the

Stirling's approximation

Stirling's approximation is a factorial approximation technique. Its formula is given as:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Stirling's approximation for n!

middle. This means that all the combinations of size n/2 need to be checked. The amount of subsets is the combination of n with k: (n!/k!(n-k!)).

With the help of Stirling's approximation, this can be reduced into:

n! ≈ (√(2πn) * (n/e)^n)

k! ≈ (√(2πk) * (k/e)^k)

(n-k)! ≈ (√(2π(n-k)) * ((n-k)/e)^(n-k))

Plugging in these values and simplifying the expression results in this:

≈ 2^(n/2) * **(√(n) * √(2π))**.

The bold ones are constant so the time complexity is O(2^(n/2)) in the worst case. Therefore, the algorithm is exponential and it is not efficient.

b) Heuristic Algorithm

We found a greedy heuristic algorithm that attempts to solve the k-clique problem. It is based on the greedy algorithm of Grimmett-McDiarmid. We found multiple sources for this and created a C++ implementation accordingly:

https://yuvalfilmus.cs.technion.ac.il/Papers/GMD.pdf

https://people.duke.edu/~jx77/Lecture_2.pdf

Here is the pseudo code version of the algorithm:

```
bool greedy_clique(int k) {
   if (clique size is 1) {
      the vertex itself is a clique by definition, return true;
   }
   vector<int> clique; //an empty vector to push the vertices that form a clique
   vector<int> vertices; //to keep track of vertices
   //fill the vertice vectır accordingly to the size of the graph
   for (all vertices in the graph) {
      vertices.push_back(i); //add it to the vector
   }
   for (all the vertices in the graph) {
      clique.clear(); //clear the previous attempt
      clique.push_back(vertices[i]); //start the clique from vertice i
      for (v in vertices) { //for all other vertices
         if (v is already included in the clique) {
            continue; //skip this vertice
         }
         bool isNext = true; //to keep track whether there isn't an edge
         for (all vertices in the clique) {
            if (if there is NOT a corresponding edge in the graph ){
               isNext = false; //don't add this vertice to the clique
               break; //break since no need to consider the rest
            }
         }
         if (isNext //it has an edge with all other vertices in the clique) {
            clique.push_back(v); //add it to the clique
            if (clique size is met) {
               return true;
            }
         }
      }
   }
   otherwise return false;
}
```

Explanation of the algorithm:

The algorithm is not an approximation algorithm, it is a greedy one. It iterates through all vertices and tries to find a clique starting with that vertice. It then iterates through all the vertices that were not added to the clique. If the current iterated vertice has a connection to all the other vertices that were added to the clique, it gets added in. Whenever the size criteria is met, it returns true. If it never meets that criteria it returns false.

In the algorithm with each step, the first adjacent vertex (that has a connection with all the previously inserted vertices inside the clique vector) is added into the clique. This is a local optimal choice since it only considers the previously inserted vertices and not the remaining unchecked ones.

The algorithm does not use any design techniques, it is a greedy one and the reason for that is to trade correctness for speed. It provides a simpler and more efficient way since it does not consider all subsets. We used this heuristic since the correct brute force implementation is computationally more expensive. Even though it doesn't always give the correct results it is faster than the brute force algorithm we previously showed.

3- Algorithm Analysis

a) Brute Force Algorithm:

Theorem: The brute force algorithm for clique problem successfully points out if there is a clique of given size k.

Proof by induction: To prove that this algorithm correctly finds if there is a clique of size k, we can use mathematical induction.

Base step:

When k = 2, the algorithm exhausts all the subsets of size 2 and checks whether there is an edge between them. If there is, it prints out the resulting clique and if not, it won’t print anything. Since all the subsets are checked, we can say that the algorithm works correctly.

Inductive step:

By the nature of the proof by induction, if F(n) is true, then F(n+1) is also correct. Therefore, we first assume that the algorithm works correctly for finding the cliques of size k where k >= 2. This assumption tells us that in the current iteration, inside the **store** array, we have k elements that are all connected to one another. At this step the algorithm will simply call itself again (**findCliques**) with the recursion. The algorithm will then look for vertices that weren't checked before as the starting parameters are updated, it will then try vertices one by one, calling the **isClique** function. isClique function will check the connection between the newly inserted element with all the present ones. If it returns true, it confirms that there is an edge between the new vertex and all the others, so the new vertex is added to the array. Since we now have k+1 elements in the store array, it simply prints the clique. If isClique returns false for all the candidates, nothing is printed on the screen and the algorithm is terminated. In both cases, the algorithm will correctly return the answer of the problem.

Worst case time complexity:

In the brute force algorithm, as explained above, stopping execution if the candidate set cannot form a clique reduces the search time and space complexity, which tries to make the algorithm more efficient. In the best case, if the algorithm finds a clique of size k earlier, then it terminates. However, in the worst case, the findCliques function has to make the maximum number of iterations and execute over all of the possible subsets of size k. Since, the number of subsets of a set is calculated by the combinations of (n choose k), where n is the size of the set and k is the size of the subset. In the worst case scenario, k will be n/2 and as explained in step 2, using Stirling's approximation, the worst case time complexity becomes O(2^(n/2)) which is terrible if the size of the graph is large.

If k is smaller/bigger than n/2, the algorithm will terminate faster as there are less subsets to check. Since k is a variable, it effects the running time of the algorithm therefore we cannot give a tight bound

.

Space complexity:

Considering the space complexity, we are only using an array to store the clique of size n where n is the amount of vertices and an adjacency matrix of n x n. Since size of the graph dominates, the complexity is simply O(n^2).

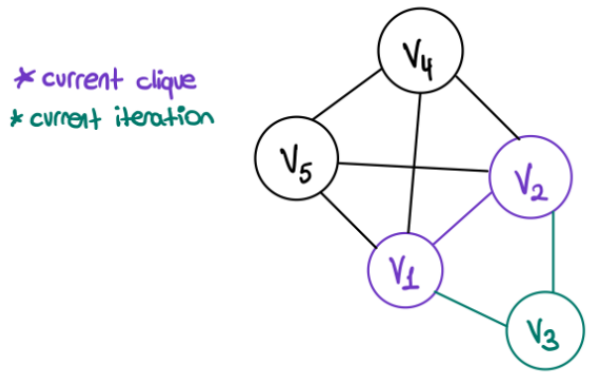
b) Heuristic Algorithm

PS: For the completeness of the report, we used n as the vertex count as previously mentioned. As this algorithm is a heuristic one, we cannot prove its correctness. The analysis is described below:

Problem = The aim of this algorithm is to determine whether there is a clique of size k in a given problem.

Heuristic Approach = The problem uses a greedy method and constructs a clique iteratively. The factor that makes this algorithm a greedy one is that it always makes locally optimal choices and not the globally optimal ones. As a result, it may not always give the best overall solution and may not always be correct.

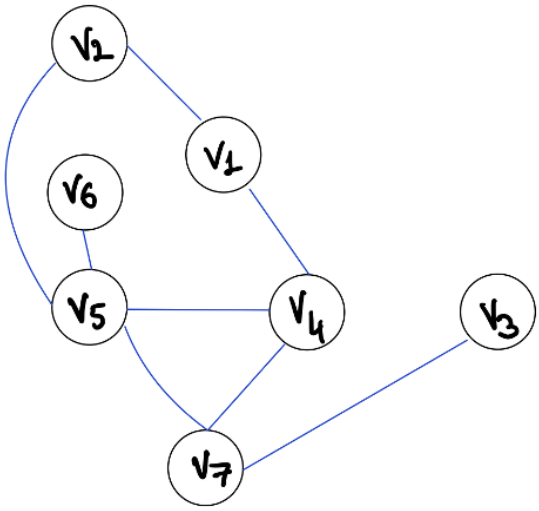
To demonstrate the limitations:



As displayed above, the choice of adding the vertice v3 is locally optimal as it matches with previously added ones. However if we are looking for a clique of size 4 in this example, the current iteration will terminate as it now included v1,v2,v3 on a local optimal choice, even though it could have found a clique of size 4 with v1,v2,v4,v5.

The reason is that since the vertices are considered by order, if a graph's first vertices are less connected. The algorithm may overlook the remaining vertices which might have achieved the clique we are looking for.

Here is an example of a false negative result:



Iteration 1 => v1,v2 break
Iteration 2 => v2, v1 break
Iteration 3 => v3, v7 break
Iteration 4 => v4, v1 break
Iteration 5 => v5, v2 break
Iteration 6 => v6, v5 break
Iteration 7 => v7, v3 break

As displayed above it failed to find the clique of size 3 even though there is one. In iterations 4,5,7 since the algorithm goes through vertices by order, it took the local optimum vertices while it could have taken the others thereby missing the clique. If the ordering was different, it could have gotten a correct result.

Hence this algorithm does not guarantee finding a clique of size k at all times. It may return false if a condition like above happens. It is limited as it does not consider all possible subsets of vertices, instead it goes by vertex order and local optimal values thereby limiting the correctness.

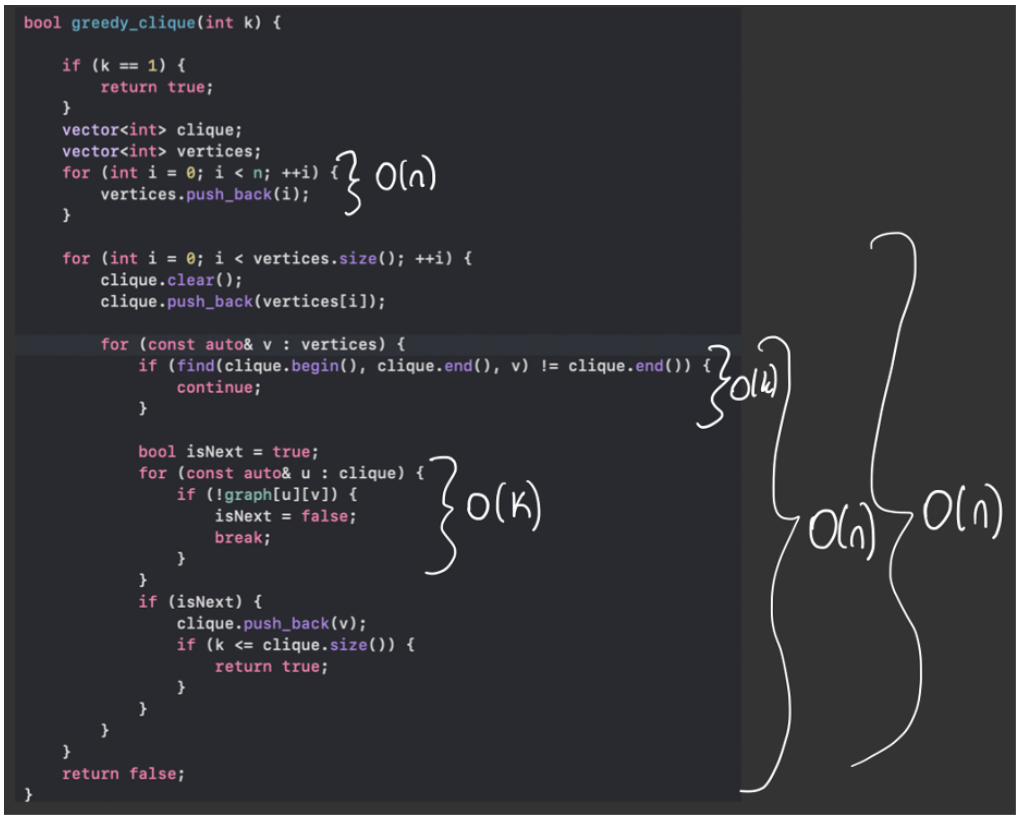
Further explanation on the correctness =

The result of the algorithm is correct when it returns true. Since it checks the existence of edges between all the vertices that are added to the clique vector, the vector will always be a clique. So when the size criteria is met, and since all vertices in the vector are connected to each other it will be a correct result.

However when the algorithm returns false, it does not prove the absence of a k clique. The algorithm does not explore all possible combinations systematically, thereby it may miss a possible clique.

Performance: The algorithm does not guarantee the global optimal solution in all cases, however it may give correct results for some types of graphs and it is more efficient than the brute force one. This will be explored in the following sections.

Time Complexity:



Creating the vertice vector has a time complexity of O(n) since it iterates from 0 to n.

The main loop will iterate for all the vertices and since there are n vertices, its complexity is O(n).

To add vertices to the candidate clique it again iterates through all the vertices in the graph, giving it the complexity O(n).

The find function checks whether the current vertice is in the candidate clique which takes O(k) times as the clique can be at most of size k.

Similarly checking the adjacency between the current vertex and the members of the clique vector takes the complexity of O(k).

Combining all of these, the worst-case time complexity of the algorithm is O(n*n * (k + k)) = **$O(n^2 * k)$.**

Space complexity:

Considering the space complexity, similarly to the previous algorithm, the size of the adjacency graph dominates, hence the complexity is O(n^2).

As a result:

Compared to the brute force algorithm it is much more efficient as this algorithm runs in polynomial time while the other one is exponential. So it gets a result much faster especially for larger sized inputs.

4- Sample Generation (Random Instance Generator)

Again we found a generative code from GeeksForGeeks = https://www.geeksforgeeks.org/how-to-create-a-random-graph-in-c/

Note = The above algorithm wasn't giving the results as an adjacency matrix and it also didn't start the vertice numbers from 0. In order to fix that we made some minor modifications.

Since the goal is to simply generate an undirected graph randomly, the parametric we are using for this is the number of edges and vertices the graph has.

While calling this function we created random numbers in the main program to be the edge count and vertex count.

For the vertices we set it to be between 7 and 12 to make the clique function easier to test. However the amount of edges should be created according to the number of vertices even though it is random. So the maximum amount of edges it may have is to make connections between all vertices which is n*(n-1)/2 while the minimum is zero. For randomness, we used the randgen library.

```
n = gen.RandInt(7,12); //random between 7 and 12 simply
cout << "The graph has " << n << " vertices" << endl;
e = gen.RandInt(0,((n * (n - 1)) / 2));
cout << "and has " << e << " edges." << endl;
```

Below is the function we are using to generate the graph, it takes the number of edges and the number of vertices as its inputs.

It first initializes an edge array and stores the edges in terms of vertices.
An example is shown below:
edge[6][0] = 1
edge[6][1] = 2

This simply means that edge number 6 is between the vertices 1 and 2.

```
void GenRandomGraphs(int NOEdge, int NOVertex)
{
   RandGen gen;
   int i, j, edge[NOEdge][2], count;
```

Then it sets all elements in the graph matrix to be zero and does the same thing for the degree array, which is used to store the degree of vertex i.

```
   // Initialize the adjacency matrix to false
   for (i = 0; i < NOVertex; i++) {
      for (j = 0; j < NOVertex; j++) {
         graph[i][j] = 0;
         d[i] = 0;
      }
   }
   i = 0;
```

After that it iterates through all edges the graph should have and tries to pick the vertices randomly. If it picks the same thing for both edges, it tries again and generates a new random number. Or if those vertices already had an edge, it again generates a new number.
Then it changes the corresponding entries in the adjacency matrix to be 1. Similarly, it increments the degree array by 1 for both vertices of that edge.

```
   // Assign random values to the number
   // of vertex and edges of the graph,
   // Using rand().
   while (i < NOEdge) {
      edge[i][0] = gen.RandInt(1,INT_MAX) % NOVertex + 1;
      edge[i][1] = gen.RandInt(1,INT_MAX) % NOVertex + 1;
      // Check if edge already exists
      if (edge[i][0] == edge[i][1]) {
         continue;
      } else if (graph[edge[i][0]-1][edge[i][1]-1] || graph[edge[i][1]-1][edge[i][0]-1]) {
         continue;
      } else {
         // Add edge to adjacency matrix
         graph[edge[i][0]-1][edge[i][1]-1] = true;
         graph[edge[i][1]-1][edge[i][0]-1] = true;
         d[edge[i][0]-1] += 1; // increment degree of first vertex of the edge
         d[edge[i][1]-1] += 1; // increment degree of first vertex of the edge
         i++;
```

```
    }
}
```

After generating, it simply loops through the adjacency matrix and prints it to the console.

```
// Print the adjacency matrix
cout << "The adjacency matrix for the generated random graph is:" << endl;
for (i = 0; i < NOVertex; i++) {
    for (j = 0; j < NOVertex; j++) {
        cout << graph[i][j] << " ";
    }
    cout << endl;
}
```

To make it easier for myself to understand I also made it print in another format. Which shows the connections of each vertex in number format like below:
1 => {2,4,6} //simply means vertex 1 has a connection with 2,4 and 6

```
cout << "----------------------" << endl;
for (i = 0; i < NOVertex; i++) {
        count = 0;
        cout << "\t" << i<< "-> { ";
        for (j = 0; j < NOEdge; j++) {
            if (edge[j][0] == i + 1) {
                cout << edge[j][1]-1 << " ";
                count++;
            }
            else if (edge[j][1] == i + 1) {
                cout << edge[j][0]-1 << " ";
                count++;
            }
            else if (j == NOEdge - 1 && count == 0)
                // Print “Isolated vertex”
                // for the vertex having
                // no degree.
                cout << "Isolated Vertex!";
        }
        cout << " }" << endl;
    }
```

```
}
```

5- Algorithm Implementations

Note: Complete code that also includes the driver (main) function is at **Appendix 3…**

a) Brute Force Algorithm

As mentioned in part 2a, for finding the cliques we found the code online, however again we had to make some minor changes to it. The online version didn't use the 0th indexes of the adjacency matrices so we modified it to make it consider those as well.

Below is the complete code we used for testing the results, it includes the code for finding the cliques:

```
#include <iostream>
#include <cmath>
#include "randgen.h"
using namespace std;

const int MAX = 100;

// Stores the vertices
int store[MAX], n;
// Graph
int graph[MAX][MAX];
// Degree of the vertices
int d[MAX];

// Function to check if the given set of vertices
// in store array is a clique or not
bool is_clique(int b)
{
   // Run a loop for all the set of edges
   // for the select vertex
   for (int i = 0; i < b; i++) {
      for (int j = i + 1; j < b; j++)
```

```
            // If any edge is missing
            if (graph[store[i]][store[j]] == 0)
                return false;
        }
        return true;
}

// Function to print the clique
void print(int n)
{
        for (int i = 0; i < n; i++)
            cout << store[i] << " ";
        cout << ", ";
}

// Function to find all the cliques of size s
void findCliques(int i, int l, int s)
{
        // Check if any vertices from i+1 can be inserted
        for (int j = i; j <= n - (s - l) + 1; j++)

            // If the degree of the graph is sufficient
            if (d[j] >= s - 1) {
                // Add the vertex to store
                store[l] = j;

                // If the graph is not a clique of size k
                // then it cannot be a clique
                // by adding another edge
                if (is_clique(l + 1)){
                    // If the length of the clique is
                    // still less than the desired size
                    if (l + 1 < s)
                        // Recursion to add vertices
                        findCliques(j, l + 1, s);
                    // Size is met
                    else
                        print(l + 1);
                }
            }
}
```

Testing the code:

We have tested 22 different samples that are generated randomly. We did not have any failures as we integrated both codes together as I also mentioned above.
We set the random parameters to make it easier to test. The number of vertices change between 7 and 12 and the cliques we are looking for can be between size 2 and 5.

Inside the sample runs we printed the related information about the random graph and the size of clique we were looking for. Also note that the problem is a decision problem but the code we are providing, finds all the cliques and prints them. This makes it a lot easier to test by checking the printed cliques with the connections in the adjacency graph. We can also change the print condition inside findClique function to simply return true, it wouldn't affect the algorithm.
All the sample runs of the above program are given below, with their related information. It is safe to say that our algorithm worked 22/22 times.

Sample Runs are at Appendix 1….

b) Heuristic Algorithm

Below is the C++ implementation of the previously mentioned algorithm:

```
bool greedy_clique(int k) {

  if (k == 1) {
    return true;
  }
  vector<int> clique;
  vector<int> vertices;
  for (int i = 0; i < n; ++i) {
    vertices.push_back(i);
  }

  for (int i = 0; i < vertices.size(); ++i) {
    clique.clear();
    clique.push_back(vertices[i]);
    for (const auto& v : vertices) {
      if (find(clique.begin(), clique.end(), v) != clique.end()) {
        continue;
      }
```

```
            bool isNext = true;
            for (const auto& u : clique) {
                if (!graph[u][v]) {
                    isNext = false;
                    break;
                }
            }
            if (isNext) {
                clique.push_back(v);
                if (k <= clique.size()) {
                    for(int x=0; x < clique.size(); x++){
                        cout << clique[x] << " ";
                    }
                    cout << endl;
                    return true;
                }
            }
        }
    }
    return false;
}
```

Testing the code:

Since the brute force algorithm is proven to be correct at all times, we used its results in order to check the validity of the heuristic one. We generated 15 samples and in 2 of them the algorithm failed to find k cliques even though there was one. So in 15 samples it has an error rate of %13. However since these sample runs are random this is not a good way to analyze the error rate, the exploratory analysis results will give us a more fitting average for the error rates. An important point is that we can also observe that in the cases that it returned true, the result was always correct, but in the ones it said no, it failed to identify the existing cliques. As a result, our algorithm suffers from false negatives and never from false positives. (mentioned in previous sections).

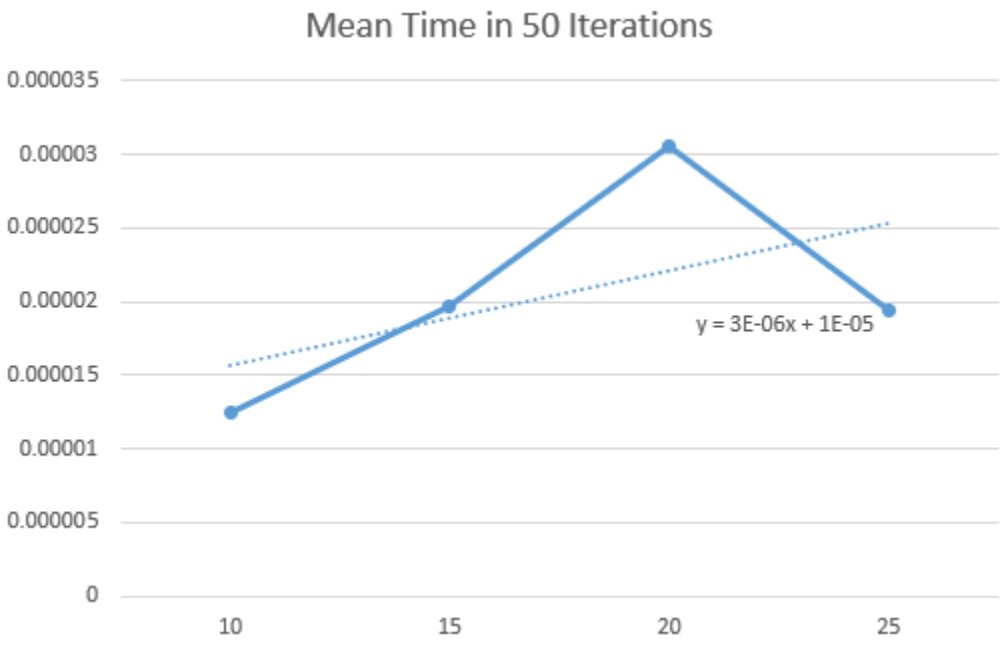
Sample Runs are at Appendix 2….

6. Experimental Analysis of The Performance (Performance Testing)

In this section, we analyzed the running time of the greedy heuristic algorithm and tried to come up with a regression equation whose upper bound is the same as the time complexity of the algorithm that is found in Section 3.b. To make sure that the input size is directly effective on the running time, we fixed the clique size (k) to 4. Then, we collected the run time data with 4 different graph sizes and 4 different execution times. The results of experiments are below and for further details you can see the appendix files and the "Experimental_Analysis.xlsx" file.

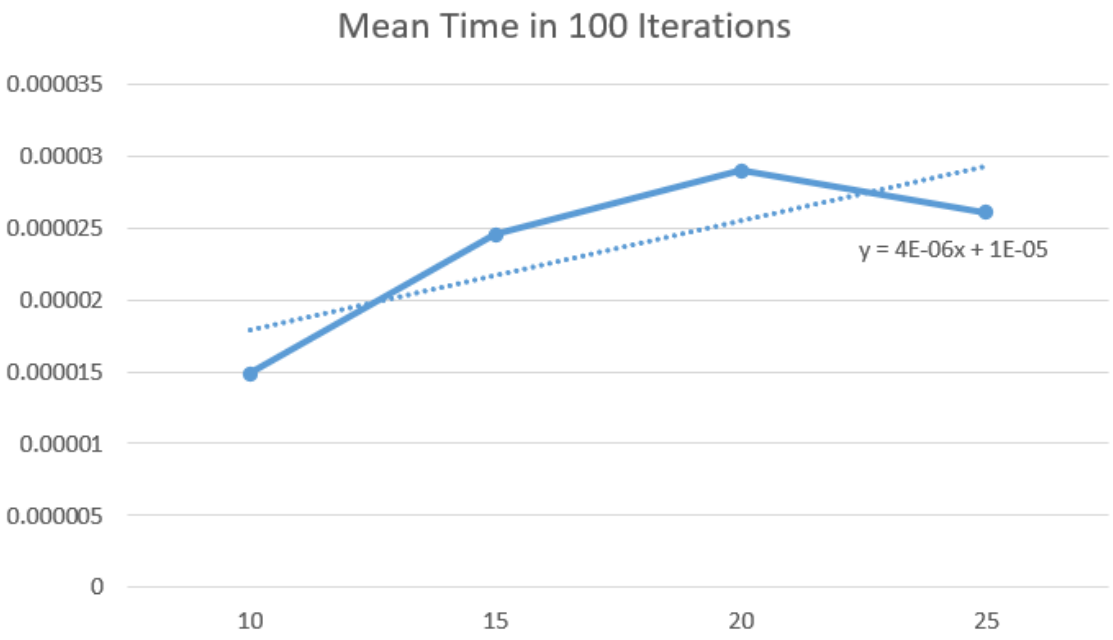
Firstly, we executed our script 50 times for each input size;

50 Runs	k=4			
Input Size	Mean Time	Standard Deviation	Lower Limit of 95% C.L.	Upper Limit of 95% C.L.
10	1.24259E-05	4.66679E-06	1.11323E-05	1.37194E-05
15	1.96459E-05	1.1231E-05	1.65329E-05	2.27589E-05
20	3.06348E-05	2.2705E-05	2.43414E-05	3.69282E-05
25	1.94428E-05	2.06864E-05	1.3709E-05	2.51767E-05



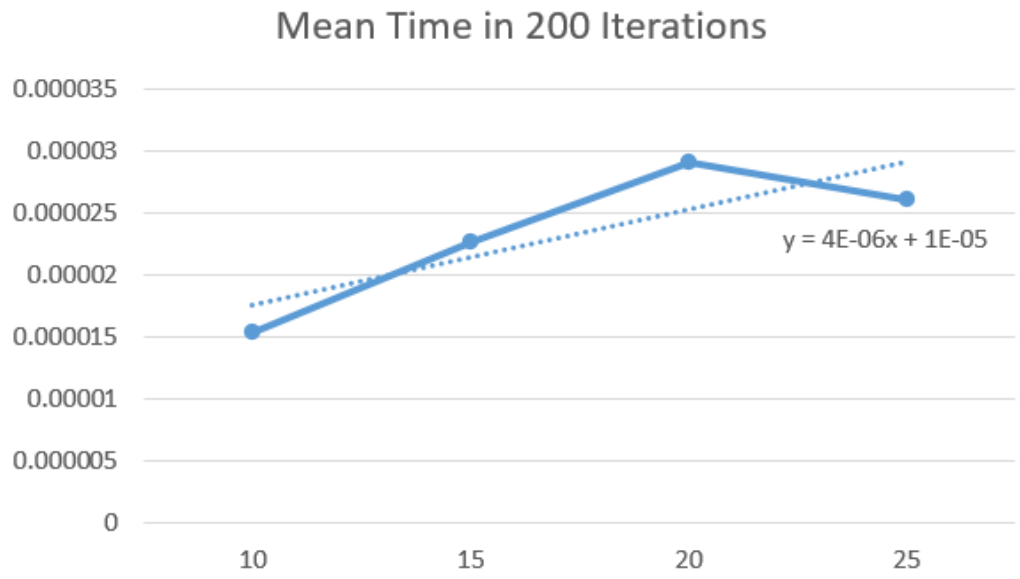
Then, we executed 100 trials for each input size;

100 Runs	k=4			
Input Size	Mean Time	Standard Deviation	Lower Limit of 95% C.L.	Upper Limit of 95% C.L.
10	1.48938E-05	5.31149E-06	1.38528E-05	1.59349E-05
15	2.45567E-05	1.53559E-05	2.1547E-05	2.75664E-05
20	2.89947E-05	2.38696E-05	2.43163E-05	3.3673E-05
25	2.60629E-05	2.53213E-05	2.11E-05	3.10258E-05



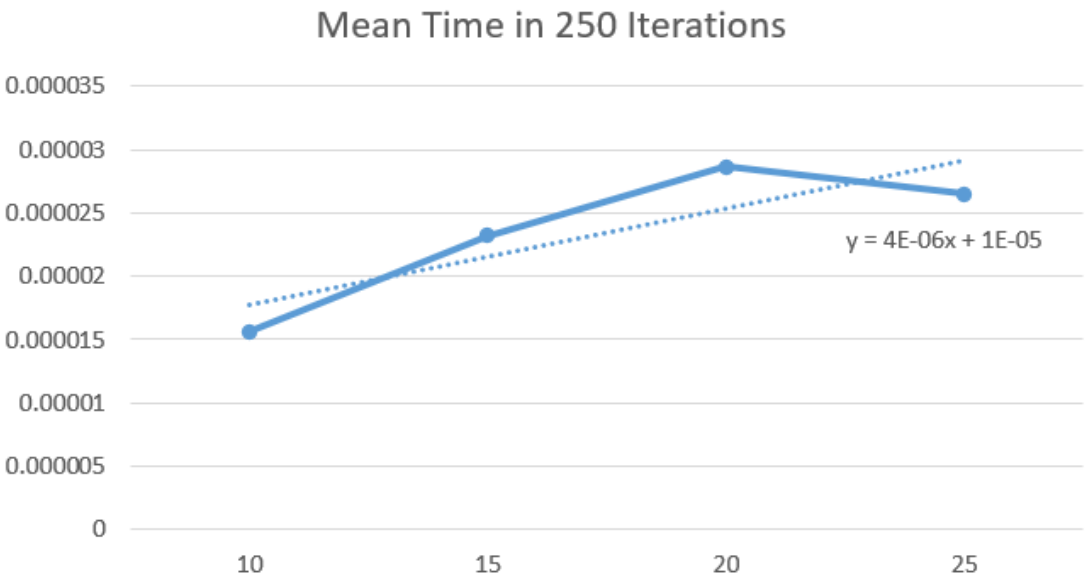
Then, we executed 200 trials for each input size;

200 Runs	k=4			
Input Size	Mean Time	Standard Deviation	Lower Limit of 95% C.L.	Upper Limit of 95% C.L.
10	1.53791E-05	6.09114E-06	1.41853E-05	1.65729E-05
15	2.26714E-05	1.36682E-05	1.99925E-05	2.53503E-05
20	2.9045E-05	2.51298E-05	2.41196E-05	3.39703E-05
25	2.60996E-05	2.52564E-05	2.11494E-05	3.10498E-05



Then, we executed 250 trials for each input size;

250 Runs	k=4			
Input Size	Mean Time	Standard Deviation	Lower Limit of 95% C.L.	Upper Limit of 95% C.L.
10	1.56557E-05	6.04319E-06	1.44713E-05	1.68402E-05
15	2.31698E-05	1.40642E-05	2.04133E-05	2.59264E-05
20	2.86603E-05	2.09972E-05	2.45449E-05	3.27756E-05
25	2.64589E-05	2.74878E-05	2.10714E-05	3.18464E-05



In each execution, we collected the run time for each inner execution. For instance, in a 50 trial execution, we collected every single trial run time and formed a sample. Then, we calculated the mean and standard deviation of the sample. To come up with an approximate running time for certain input sizes, which we determined as 10, 15, 20 and 25, we formed 95 percent confidence intervals for minimum and maximum running times.
Finally, we formed a regression line and derived a running time expression based on that equation.

$y = 4*(10^{-6}) x + 1*10^{-5}$; where y is the running time and x is the input size

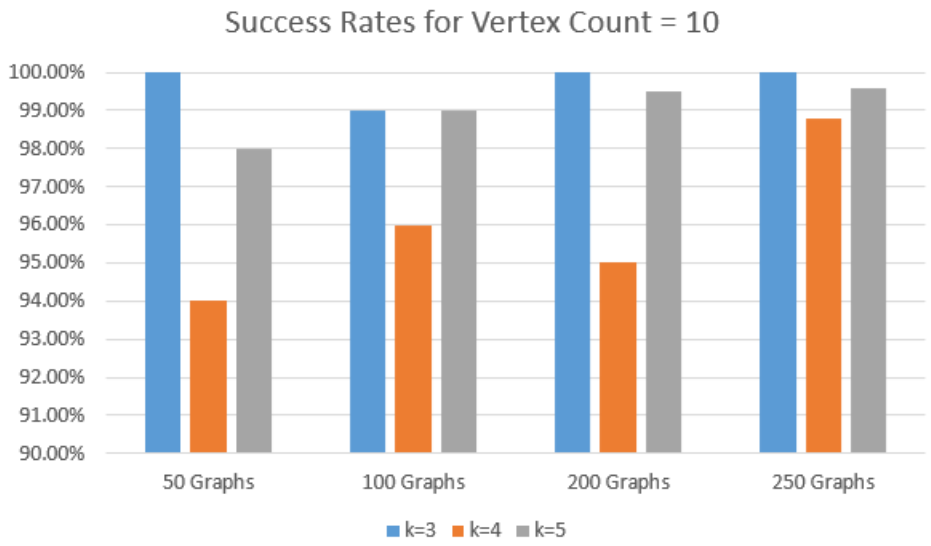
It can be inferred from the equation that the theoretical analysis that is conducted in Section 3-b is correct and the average run time of the greedy heuristic algorithm is in O(n).

7. Experimental Analysis of the Quality

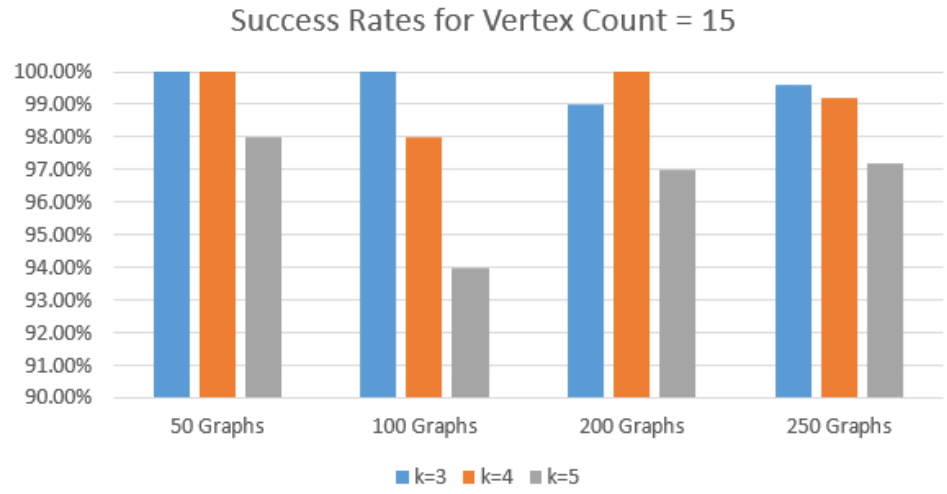
In this section, we analyzed the quality of the greedy heuristic algorithm and tried to come up with a regression equation which denotes the relationship between the quality of the heuristic algorithm and the graph size. To run our experiments, we used the brute force algorithm as the base and exact algorithm, and we compare the results that are returned by the heuristic algorithm with the results that are returned by the brute force algorithm. By trying different combinations of graph sizes and clique sizes, and executing them multiple times, we formed samples and analyzed them. Below, you can see the results of the experiments and the conclusive explanations.

Input Size (V) = 10				
Success Rates	50 Graphs	100 Graphs	200 Graphs	250 Graphs
k=3	100.00%	99.00%	100.00%	100.00%
k=4	94.00%	96.00%	95.00%	98.80%
k=5	98.00%	99.00%	99.50%	99.60%

*Success rates when the input size is 10 and different size of cliques is searched



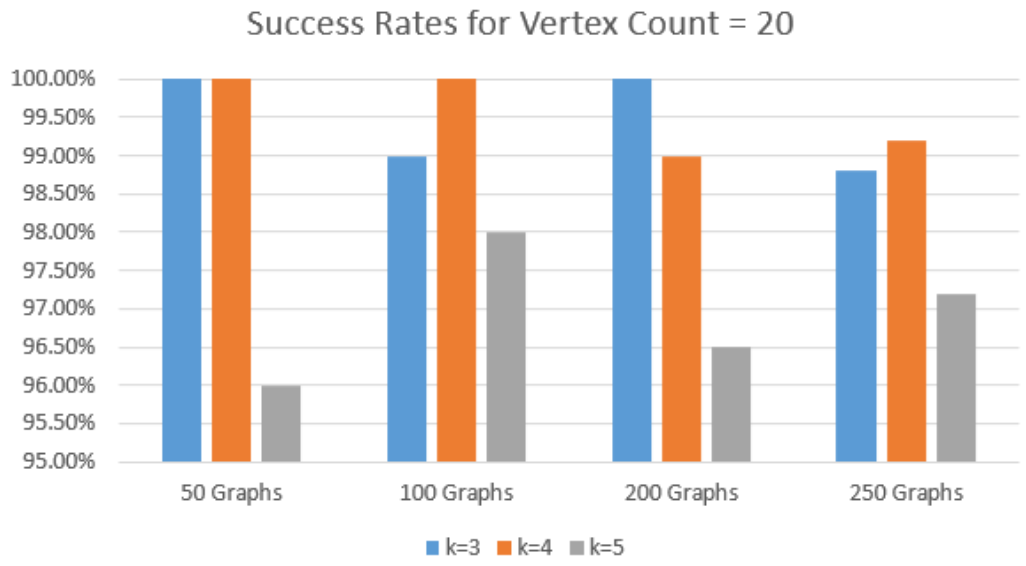
Input Size (V) = 15				
Success Rates	50 Graphs	100 Graphs	200 Graphs	250 Graphs
k=3	100.00%	100.00%	99.00%	99.60%
k=4	100.00%	98.00%	100.00%	99.20%
k=5	98.00%	94.00%	97.00%	97.20%



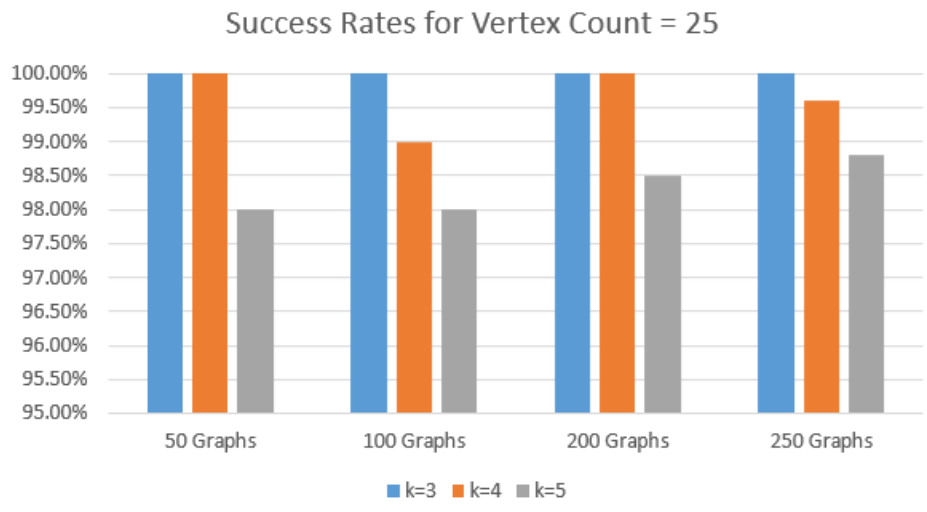
*Success rates when the input size is 15 and different size of cliques is searched

Input Size (V) = 20				
Success Rates	50 Graphs	100 Graphs	200 Graphs	250 Graphs
k=3	100.00%	99.00%	100.00%	98.80%
k=4	100.00%	100.00%	99.00%	99.20%
k=5	96.00%	98.00%	96.50%	97.20%

*Success rates when the input size is 20 and different size of cliques is searched



Input Size (V) = 25				
Success Rates	50 Graphs	100 Graphs	200 Graphs	250 Graphs
k=3	100.00%	100.00%	100.00%	100.00%
k=4	100.00%	99.00%	100.00%	99.60%
k=5	98.00%	98.00%	98.50%	98.80%

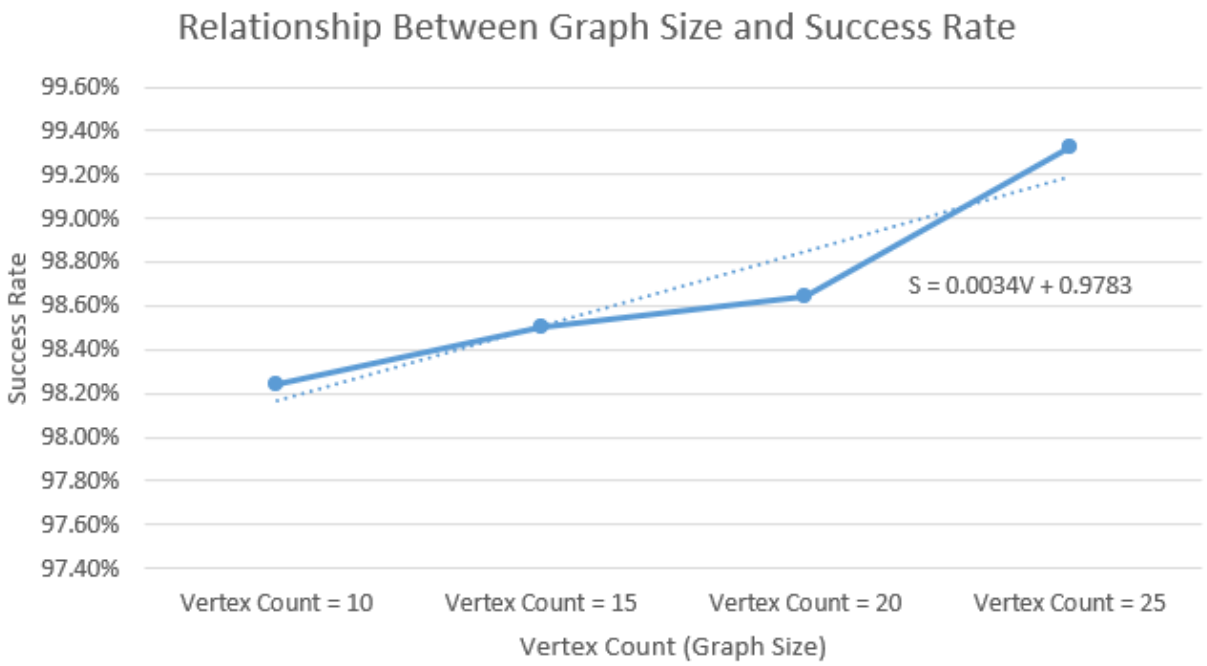


*Success rates when the input size is 25 and different size of cliques is searched

After analyzing all the samples collectively, we realized that regardless of the clique size, as the graph size increases, the success rate of the heuristic algorithm increases as well.

Summary				
	Vertex Count = 10	Vertex Count = 15	Vertex Count = 20	Vertex Count = 25
Success Rates	98.24%	98.50%	98.64%	99.33%

* In this table, success rates are calculated by taking the arithmetic mean of all success rates under the responsive vertex count.



***Where S is the success rate and V is the graph size.

When all the experiments are considered, it can be concluded that the findings support the idea that we put forward on Section 3-b. In most of the cases, the algorithm returns true as it finds a k-size clique and in some cases it returns a false negative when there actually exists a clique. Therefore, the heuristic algorithm works approximately with a 100% success rate. In addition, as the graph size increases, the success rate of the heuristic algorithm increases as well regardless of the clique size. It is not surprising at all because of the fact that finding a clique becomes more likely as the graph size increases.

8. Experimental Analysis of the Correctness (Functional Testing)

To check the correctness of the implementation of the heuristic algorithm, we first applied black box testing, in which we considered the functional specifications and requirements of the algorithm. To achieve this, first we formed equivalence classes. The followings are the considered conditions and test cases;

Section 8: Functional Testing	
Equivalence Classes	
Valid Classes	Invalid Classes
A graph of size larger than 1 (with no clique) and k>1	Vertex count < 1
A graph of size larger than 1 (with clique(s)) and k>1	An empty graph and k>1

1-) Input: k > 1 and graph is empty

Expected output: **false ;** actual output: **false**.

This test case checks if the function handles the scenario of an empty graph correctly.

2-) Input: a graph with -1 vertices

Expected output: **error message;** actual output: **error message**

In this test case, an invalid graph is tried to be given as an input, and as expected function returned with an error

3-) Input: k > 1 and a non-empty graph (in which there is no clique of size k)

Expected output: **false ;** actual output: **false**.

In this test case, we checked if the function returns false when there is no clique of size k.

4-) Input: k > 1 and a non-empty graph (in which there is a clique of size k)

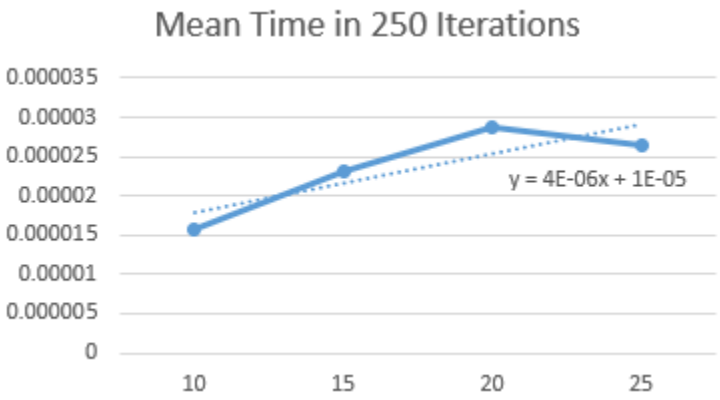
Expected output: **true ;** actual output: **(depends but mostly true)**.

In this test case, we checked if the function returns true when there is a clique of size k. As it is put forward in the Section 7, in most of the trials, function returns true as expected, but in some cases it returns false negative.

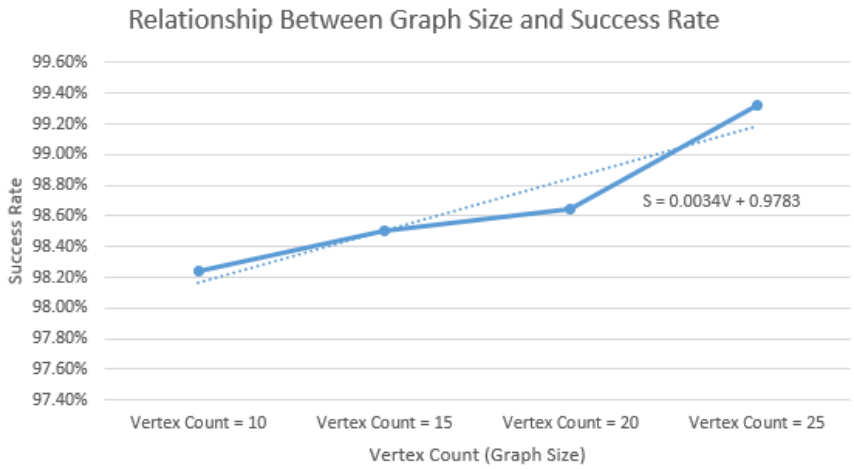
9. Discussion

In conclusion, we have tried to understand the k-clique problem and solve it by using two different algorithms. The brute force algorithm gives an exact result but in an inefficiently longer time, whereas the heuristic algorithm gives an approximately exact result with some defects. However, after all the experimental process it can be said that the better way to approach the k-clique problem is the heuristic one. In addition, we also experimented the theoretical analysis and saw that the time complexity and correctness of the heuristic algorithm is coherent with our findings. One can see all the used tools and techniques in the appendix part and the shared files.

Performance of the Heuristic Algorithm



Quality of the Heuristic Algorithm



Appendix 1: (Sample runs for Brute Force)

```
Random graph generation:
The graph has 10 vertices
and has 31 edges.
The adjacency matrix for the generated random graph is:
0 0 1 0 1 0 0 0 1 1
0 0 0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1 1 1
0 0 1 0 0 0 0 1 1 1
1 1 1 0 0 1 1 0 1 1
0 1 1 0 1 0 1 1 1 1
0 1 1 0 1 1 0 0 1 1
0 1 1 1 0 1 0 0 0 1
1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 1 0 0
----------------------
    0-> { 4 9 2 8  }
    1-> { 6 9 8 4 7 5  }
    2-> { 5 9 3 8 4 7 6 0  }
    3-> { 7 2 9 8  }
    4-> { 8 6 0 2 5 9 1  }
    5-> { 2 9 7 6 4 8 1  }
    6-> { 1 4 5 8 2 9  }
    7-> { 3 5 9 2 1  }
    8-> { 4 2 6 1 5 0 3  }
    9-> { 5 2 1 7 3 0 4 6  }

----------------------------------------
Looking for cliques of size: 4........
0 2 4 8 , 0 2 4 9 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 8 vertices
and has 16 edges.
The adjacency matrix for the generated random graph is:
0 1 0 0 1 0 1 1
1 0 0 1 0 1 1 0
0 0 0 0 1 0 0 1
0 1 0 0 1 1 0 1
1 0 1 1 0 1 1 0
0 1 0 1 1 0 1 0
1 1 0 0 1 1 0 1
1 0 1 1 0 0 1 0
----------------------
    0-> { 4 7 6 1  }
    1-> { 5 6 3 0  }
    2-> { 4 7  }
    3-> { 7 4 1 5  }
    4-> { 3 5 0 2 6  }
    5-> { 1 6 4 3  }
    6-> { 5 7 1 4 0  }
    7-> { 3 6 0 2  }

----------------------------------------
Looking for cliques of size: 2........
0 1 , 0 4 , 0 6 , 0 7 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 11 vertices
and has 29 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 1 0 0 1 1 1 0
1 0 1 0 1 1 0 1 0 0 1
1 1 0 1 0 1 0 0 0 0 0
1 0 1 0 0 1 0 0 0 0 1
1 1 0 0 0 1 1 1 1 1 1
0 1 1 1 1 0 0 1 1 1 1
0 0 0 0 1 0 0 1 1 0 0
1 1 0 0 1 1 1 0 0 0 1
1 0 0 0 1 1 1 0 0 0 0
1 0 0 0 1 1 0 0 0 0 0
0 1 0 1 1 1 0 1 0 0 0
----------------------
    0-> { 4 7 9 1 3 2 8  }
    1-> { 4 0 10 2 7 5  }
    2-> { 3 0 5 1  }
    3-> { 10 2 0 5  }
    4-> { 0 7 8 1 6 10 5 9  }
    5-> { 10 9 3 8 2 7 4 1  }
    6-> { 4 8 7  }
    7-> { 0 4 6 10 1 5  }
    8-> { 4 5 6 0  }
    9-> { 0 5 4  }
    10-> { 3 5 1 7 4  }

----------------------------------------
Looking for cliques of size: 2.......
0 1 , 0 2 , 0 3 , 0 4 , 0 7 , 0 8 , 0 9 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 9 vertices
and has 19 edges.
The adjacency matrix for the generated random graph is:
0 0 1 1 0 1 1 0 1
0 0 0 1 0 0 0 1 1
1 0 0 0 1 1 0 1 0
1 1 0 0 0 1 0 1 1
0 0 1 0 0 1 0 1 1
1 0 1 1 1 0 1 0 1
1 0 0 0 0 1 0 0 0
0 1 1 1 1 0 0 0 0
1 1 0 1 1 1 0 0 0
----------------------
    0-> { 3 6 5 2 8  }
    1-> { 7 3 8  }
    2-> { 4 7 5 0  }
    3-> { 0 7 8 1 5  }
    4-> { 7 2 8 5  }
    5-> { 8 0 6 4 2 3  }
    6-> { 0 5  }
    7-> { 4 3 2 1  }
    8-> { 5 4 3 1 0  }

----------------------------------------
Looking for cliques of size: 4........
0 3 5 8 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 10 vertices
and has 3 edges.
The adjacency matrix for the generated random graph is:
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0
1 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
------------------------
    0-> { 7  }
    1-> { Isolated Vertex! }
    2-> { Isolated Vertex! }
    3-> { Isolated Vertex! }
    4-> { 9  }
    5-> { Isolated Vertex! }
    6-> { 7  }
    7-> { 0 6  }
    8-> { Isolated Vertex! }
    9-> { 4  }

----------------------------------------
Looking for cliques of size: 5........

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 9 vertices
and has 21 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 1 1 1 0 1
1 0 0 1 1 0 1 0 0
1 0 0 0 0 1 0 1 0
1 1 0 0 0 0 1 1 0
1 1 0 0 0 1 1 0 1
1 0 1 0 1 0 1 1 1
1 1 0 1 1 1 0 0 0
0 0 1 1 0 1 0 0 1
1 0 0 0 1 1 0 1 0
----------------------
    0-> { 6 5 2 1 3 4 8  }
    1-> { 4 0 3 6  }
    2-> { 7 5 0  }
    3-> { 6 0 1 7  }
    4-> { 1 8 5 0 6  }
    5-> { 0 2 7 8 6 4  }
    6-> { 3 0 5 1 4  }
    7-> { 2 5 8 3  }
    8-> { 5 4 7 0  }

----------------------------------------
Looking for cliques of size: 2........
0 1 , 0 2 , 0 3 , 0 4 , 0 5 , 0 6 , 0 8 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 8 vertices
and has 3 edges.
The adjacency matrix for the generated random graph is:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0
-----------------------
    0-> { Isolated Vertex! }
    1-> { 6  }
    2-> { 5 6  }
    3-> { Isolated Vertex! }
    4-> { Isolated Vertex! }
    5-> { 2  }
    6-> { 1 2  }
    7-> { Isolated Vertex! }

-----------------------------------------
Looking for cliques of size: 3........

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 8 vertices
and has 25 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 0 1 1 1
1 0 1 1 1 1 1 0
1 1 0 1 1 1 1 1
1 1 1 0 1 1 0 1
0 1 1 1 0 1 1 1
1 1 1 1 1 0 1 1
1 1 1 0 1 1 0 1
1 0 1 1 1 1 1 0
-----------------------
    0-> { 1 7 3 2 5 6  }
    1-> { 3 0 4 6 2 5  }
    2-> { 6 1 7 0 3 4 5  }
    3-> { 1 5 0 7 2 4  }
    4-> { 7 1 5 6 2 3  }
    5-> { 7 3 4 1 0 6 2  }
    6-> { 2 1 4 7 5 0  }
    7-> { 5 4 0 2 3 6  }

-----------------------------------------
Looking for cliques of size: 5........
0 1 2 3 5 , 0 1 2 5 6 , 0 2 3 5 7 , 0 2 5 6 7 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 12 vertices
and has 34 edges.
The adjacency matrix for the generated random graph is:
0 0 0 1 0 1 1 1 0 0 1 1
0 0 1 1 1 0 1 0 0 0 0 1
0 1 0 0 1 0 1 1 1 0 0 1
1 1 0 0 0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0 0 0 1 1
1 0 0 1 1 0 0 0 0 1 0 1
1 1 1 1 1 0 0 0 1 0 1 1
1 0 1 0 0 0 0 0 1 0 1 0
0 0 1 0 0 0 1 1 0 0 1 0
0 0 0 1 0 1 0 0 0 0 1 0
1 0 0 1 1 0 1 1 1 1 0 1
1 1 1 0 1 1 1 0 0 0 1 0
-----------------------
    0-> { 6 5 11 10 7 3  }
    1-> { 2 4 11 6 3  }
    2-> { 1 8 7 11 6 4  }
    3-> { 10 5 9 0 1 6  }
    4-> { 6 10 1 5 11 2  }
    5-> { 0 3 4 11 9  }
    6-> { 4 8 0 11 10 2 1 3  }
    7-> { 2 10 8 0  }
    8-> { 6 2 7 10  }
    9-> { 3 10 5  }
    10-> { 4 3 6 7 9 0 11 8  }
    11-> { 6 0 2 4 1 5 10  }

-----------------------------------------
Looking for cliques of size: 3.......
0 3 5 , 0 3 6 , 0 3 10 , 0 5 11 , 0 6 10 , 0 6 11 , 0 7 10 , 0 10 11 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 10 vertices
and has 24 edges.
The adjacency matrix for the generated random graph is:
0 0 1 1 1 0 1 1 1 0
0 0 0 1 0 0 0 1 0 0
1 0 0 1 1 1 1 0 0 1
1 1 1 0 0 1 1 0 1 0
1 0 1 0 0 0 0 0 0 1
0 0 1 1 0 0 1 0 1 1
1 0 1 1 0 1 0 1 1 0
1 1 0 0 0 0 1 0 1 0
1 0 0 1 0 1 1 1 0 1
0 0 1 0 1 1 0 0 1 0
-----------------------
    0-> { 3 8 4 7 2 6  }
    1-> { 7 3  }
    2-> { 6 9 3 4 0 5  }
    3-> { 1 6 0 8 2 5  }
    4-> { 0 2 9  }
    5-> { 6 8 9 3 2  }
    6-> { 7 5 8 2 3 0  }
    7-> { 6 1 8 0  }
    8-> { 6 5 7 0 3 9  }
    9-> { 2 5 8 4  }

-----------------------------------------
Looking for cliques of size: 4........
0 2 3 6 , 0 3 6 8 , 0 6 7 8 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 12 vertices
and has 45 edges.
The adjacency matrix for the generated random graph is:
0 0 1 1 1 0 1 0 1 1 0 0
0 0 1 1 1 1 1 1 1 1 1 1
1 1 0 1 0 0 1 1 0 1 0 1
1 1 1 0 1 1 1 0 0 1 1 0
1 1 0 1 0 0 1 1 0 1 1 0
0 1 0 1 0 0 1 0 1 0 0 1
1 1 1 1 1 1 0 0 1 1 1 0
0 1 1 0 1 0 0 0 1 1 1 1
1 1 0 0 0 1 1 1 0 1 1 1
1 1 1 1 1 0 1 1 1 0 1 1
0 1 0 1 1 0 1 1 1 1 0 0
0 1 1 0 0 1 0 1 1 1 0 0
----------------------
    0-> { 9 4 8 3 6 2  }
    1-> { 4 6 7 5 2 11 3 8 10 9  }
    2-> { 11 9 3 6 1 7 0  }
    3-> { 10 6 2 9 0 1 4 5  }
    4-> { 1 6 10 0 7 9 3  }
    5-> { 11 1 8 3 6  }
    6-> { 3 4 1 10 2 0 9 8 5  }
    7-> { 10 1 2 4 8 11 9  }
    8-> { 10 9 7 0 5 11 1 6  }
    9-> { 0 2 8 3 4 10 11 1 6 7  }
    10-> { 3 7 4 8 6 9 1  }
    11-> { 2 5 1 9 8 7  }

----------------------------------------
Looking for cliques of size: 4.......
0 2 3 6 , 0 2 3 9 , 0 2 6 9 , 0 3 4 6 , 0 3 4 9 , 0 3 6 9 , 0 4 6 9 , 0 6 8 9 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 11 vertices
and has 38 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 0 1 1 0 0 1 1
1 0 0 1 0 1 0 0 1 1 1
1 0 0 1 0 1 0 1 1 1 1
1 1 1 0 1 1 0 1 1 1 1
0 0 0 1 0 1 0 1 1 1 0
1 1 1 1 1 1 0 1 1 1 0
1 0 0 0 0 1 0 1 1 1 0
0 0 1 1 1 1 1 0 1 0 1
0 1 1 1 1 1 1 1 0 0 1
1 1 1 1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 1 1 0 0
-----------------------
    0-> { 2 9 10 5 6 3 1  }
    1-> { 8 10 9 3 5 0  }
    2-> { 0 5 10 3 8 7 9  }
    3-> { 7 1 2 9 0 8 10 5 4  }
    4-> { 5 9 8 7 3  }
    5-> { 2 8 0 4 7 9 6 3 1  }
    6-> { 7 9 0 8 5  }
    7-> { 6 8 10 3 5 2 4  }
    8-> { 1 7 10 5 2 6 4 3  }
    9-> { 0 1 6 3 4 5 2  }
    10-> { 7 1 2 8 0 3  }

-----------------------------------------
Looking for cliques of size: 5........
0 1 3 5 9 , 0 2 3 5 9 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 10 vertices
and has 24 edges.
The adjacency matrix for the generated random graph is:
0 0 1 0 1 0 0 1 1 1
0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 0 0 1 1 1
0 1 0 0 1 0 1 0 1 0
1 0 1 1 0 1 1 1 0 1
0 0 0 0 1 0 0 1 1 0
0 1 0 1 1 0 0 0 1 0
1 1 1 0 1 1 0 0 0 1
1 0 1 1 0 1 1 0 0 0
1 0 1 0 1 0 0 1 0 0
-----------------------
    0-> { 7 9 2 8 4  }
    1-> { 6 2 3 7  }
    2-> { 9 1 4 0 8 7  }
    3-> { 4 1 8 6  }
    4-> { 9 2 6 7 3 5 0  }
    5-> { 4 7 8  }
    6-> { 1 4 3 8  }
    7-> { 0 9 4 5 1 2  }
    8-> { 2 3 6 0 5  }
    9-> { 2 4 0 7  }


-----------------------------------------
Looking for cliques of size: 4........
0 2 4 7 , 0 2 4 9 , 0 2 7 9 , 0 4 7 9 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 12 vertices
and has 60 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 1 1 0 1 1 1 1 1
1 0 0 1 1 1 1 1 1 1 1 1
1 0 0 1 1 1 1 1 1 1 1 1
1 1 1 0 1 1 1 1 1 1 0 1
1 1 1 1 0 1 1 1 1 1 1 0
1 1 1 1 1 0 0 1 1 1 0 1
0 1 1 1 1 0 0 1 1 1 1 1
1 1 1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 1 1 0 1 1
1 1 1 0 1 0 1 1 1 1 0 1
1 1 1 1 0 1 1 1 1 1 1 0
----------------------
    0-> { 1 7 5 4 9 10 3 11 2 8  }
    1-> { 0 3 8 9 7 10 4 6 5 11  }
    2-> { 3 6 9 5 10 4 0 8 11 7  }
    3-> { 2 1 9 7 4 5 11 0 8 6  }
    4-> { 8 6 9 7 10 3 0 5 2 1  }
    5-> { 2 9 11 3 0 4 7 8 1  }
    6-> { 2 4 7 9 8 11 10 1 3  }
    7-> { 6 4 3 0 9 1 5 11 8 2 10  }
    8-> { 10 4 11 1 6 9 2 7 5 3 0  }
    9-> { 2 4 3 10 6 5 7 1 0 8 11  }
    10-> { 8 9 11 4 0 2 6 1 7  }
    11-> { 8 10 5 3 0 6 7 2 9 1  }

----------------------------------------
Looking for cliques of size: 2........
0 1 , 0 2 , 0 3 , 0 4 , 0 5 , 0 7 , 0 8 , 0 9 , 0 10 , 0 11
Program ended with exit code: 0
```

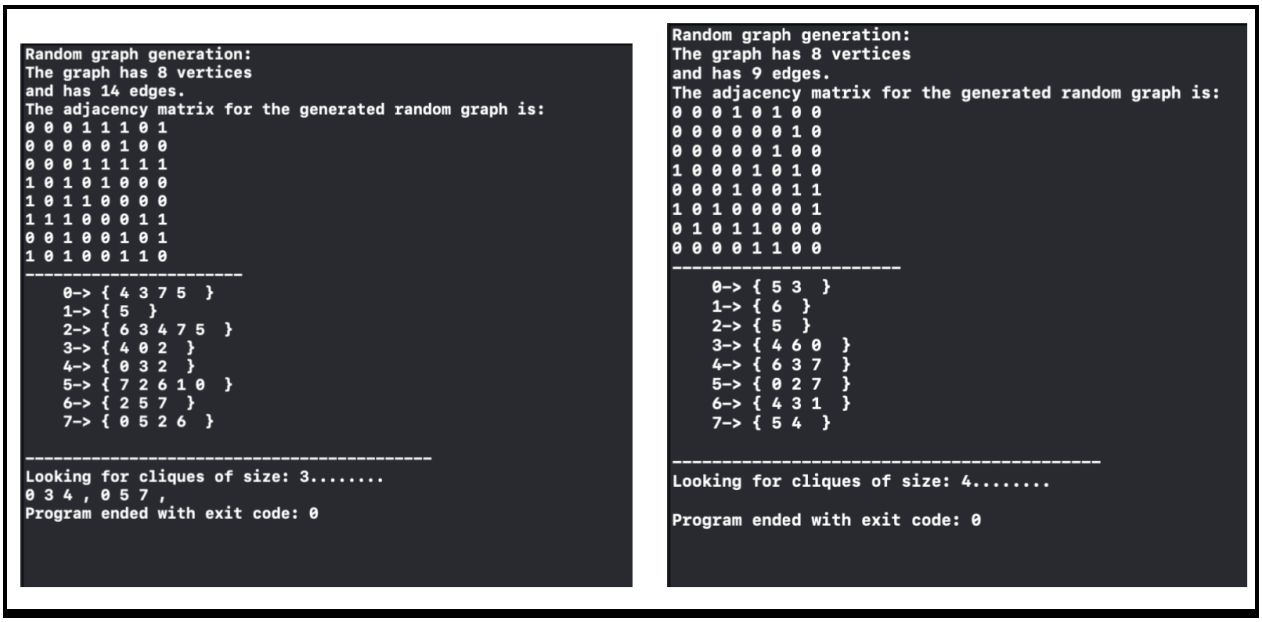
```
Random graph generation:
The graph has 7 vertices
and has 17 edges.
The adjacency matrix for the generated random graph is:
0 1 0 1 0 1 1
1 0 1 1 1 1 1
0 1 0 1 1 1 0
1 1 1 0 1 1 1
0 1 1 1 0 0 1
1 1 1 1 0 0 1
1 1 0 1 1 1 0
------------------------
    0-> { 6 1 3 5  }
    1-> { 3 0 2 4 5 6  }
    2-> { 3 4 5 1  }
    3-> { 1 5 2 4 6 0  }
    4-> { 6 2 3 1  }
    5-> { 6 3 2 0 1  }
    6-> { 5 0 4 3 1  }

-----------------------------------------
Looking for cliques of size: 2........
0 1 , 0 3 , 0 5 , 0 6 ,
Program ended with exit code: 0
```

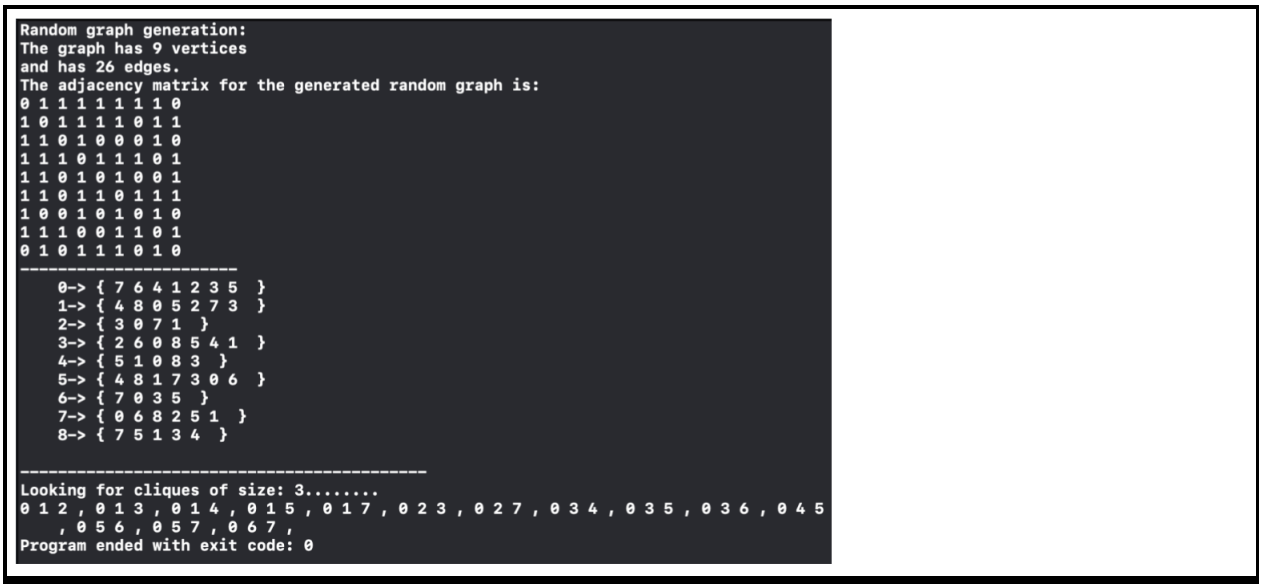
```
Random graph generation:
The graph has 12 vertices
and has 28 edges.
The adjacency matrix for the generated random graph is:
0 1 0 1 1 0 0 0 1 1 0 0
1 0 0 0 1 0 0 1 1 1 1 0
0 0 0 0 0 1 0 0 0 1 0 0
1 0 0 0 1 0 1 0 0 1 0 0
1 1 0 1 0 1 0 1 0 1 0 0
0 0 1 0 1 0 1 0 0 0 0 1
0 0 0 1 0 1 0 1 1 0 0 0
0 1 0 0 1 0 1 0 1 0 1 1
1 1 0 0 0 0 1 1 0 0 0 1
1 1 1 1 1 0 0 0 0 0 1 1
0 1 0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 1 1 0 0
----------------------
    0-> { 9 4 8 3 1  }
    1-> { 10 8 4 9 0 7  }
    2-> { 9 5  }
    3-> { 6 9 0 4  }
    4-> { 1 0 7 3 9 5  }
    5-> { 11 6 2 4  }
    6-> { 3 5 8 7  }
    7-> { 8 10 11 4 6 1  }
    8-> { 7 1 0 6 11  }
    9-> { 0 11 2 1 10 3 4  }
    10-> { 7 1 9  }
    11-> { 5 9 7 8  }

----------------------------------------
Looking for cliques of size: 3........
0 1 4 , 0 1 8 , 0 1 9 , 0 3 4 , 0 3 9 , 0 4 9 ,
Program ended with exit code: 0
```

```
The graph has 10 vertices
and has 43 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 1 1 1 1 1 1
1 0 1 1 1 0 1 1 1 1
1 1 0 1 1 1 1 1 1 1
1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1
1 0 1 1 1 0 1 1 1 1
1 1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 0 1 0
1 1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 0 1 0
----------------------
    0-> { 3 7 6 5 2 8 1 4 9  }
    1-> { 0 7 8 6 2 9 4 3  }
    2-> { 0 6 4 9 5 3 1 8 7  }
    3-> { 0 4 8 7 2 6 9 5 1  }
    4-> { 3 7 2 6 5 8 0 9 1  }
    5-> { 9 0 2 7 4 8 3 6  }
    6-> { 0 2 7 8 4 9 3 1 5  }
    7-> { 0 4 3 6 5 1 8 2  }
    8-> { 3 0 6 4 1 5 9 7 2  }
    9-> { 5 2 6 4 3 8 0 1  }

----------------------------------------
Looking for cliques of size: 3........
0 1 2 , 0 1 3 , 0 1 4 , 0 1 6 , 0 1 7 , 0 1 8 , 0 1 9 , 0 2 3 , 0 2 4 , 0 2 5 , 0 2 6
    , 0 2 7 , 0 2 8 , 0 2 9 , 0 3 4 , 0 3 5 , 0 3 6 , 0 3 7 , 0 3 8 , 0 3 9 , 0 4 5 ,
    0 4 6 , 0 4 7 , 0 4 8 , 0 4 9 , 0 5 6 , 0 5 7 , 0 5 8 , 0 5 9 , 0 6 7 , 0 6 8 , 0
    6 9 , 0 7 8 , 0 8 9 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 12 vertices
and has 12 edges.
The adjacency matrix for the generated random graph is:
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0 1 0
1 1 0 1 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 0 1 1 0
0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
----------------------
    0-> { 6 5  }
    1-> { 5 6  }
    2-> { 3  }
    3-> { 5 8 2  }
    4-> { 8 10  }
    5-> { 3 0 1  }
    6-> { 0 8 1  }
    7-> { Isolated Vertex! }
    8-> { 10 4 9 6 3  }
    9-> { 8  }
    10-> { 8 4  }
    11-> { Isolated Vertex! }

----------------------------------------
Looking for cliques of size: 2........
0 5 , 0 6 ,
Program ended with exit code: 0
```

```
Random graph generation:
The graph has 8 vertices
and has 19 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 1 0 0 0
1 0 1 0 1 1 1 0
1 1 0 1 1 1 1 0
1 0 1 0 1 1 1 1
1 1 1 1 0 1 1 0
0 1 1 1 1 0 1 0
0 1 1 1 1 1 0 0
0 0 0 1 0 0 0 0
----------------------
    0-> { 1 2 4 3  }
    1-> { 0 4 2 6 5  }
    2-> { 4 0 1 5 6 3  }
    3-> { 7 6 0 2 5 4  }
    4-> { 5 2 6 0 1 3  }
    5-> { 4 6 2 1 3  }
    6-> { 4 5 1 3 2  }
    7-> { 3  }

----------------------------------------
Looking for cliques of size: 3........
0 1 2 , 0 1 4 , 0 2 3 , 0 2 4 , 0 3 4 ,
Program ended with exit code: 0
```





Appendix 2: (Sample Runs for Heuristic Algorithm)

```
The graph has 18 vertices
and has 117 edges.
The adjacency matrix for the generated random graph is:
0 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1
0 1 0 1 1 1 1 1 1 0 0 1 1 1 0 1 1 1
0 1 1 0 1 0 1 0 1 1 1 0 0 1 1 0 1 0
1 1 1 1 0 1 1 1 1 0 0 0 0 1 1 0 1 0
1 1 1 0 1 0 1 0 1 0 1 1 0 0 1 0 1 1
1 1 1 1 1 1 0 1 0 1 1 1 1 0 0 1 0 1
1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 1 0 1
1 1 1 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1
1 1 0 1 0 0 1 1 1 0 1 1 0 1 1 0 1 0
0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1
1 1 1 0 0 1 1 0 1 1 1 0 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 0 1 1 0 1 1 1 1 0
1 1 1 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1
1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1
1 1 1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 1
1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1
1 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 1 0
----------------------
    0-> { 9 4 5 13 16 6 11 14 15 8 12 7 1 17  }
    1-> { 5 14 15 3 17 2 4 6 10 12 8 0 13 11 9  }
    2-> { 3 15 5 12 17 13 1 6 8 16 11 7 4  }
    3-> { 8 2 10 1 9 4 6 13 14 16  }
    4-> { 0 8 3 1 16 7 13 6 14 2 5  }
    5-> { 0 1 2 6 10 14 11 8 17 16 4  }
    6-> { 17 11 0 5 3 7 1 2 10 15 4 12 9  }
    7-> { 15 6 9 4 12 10 0 14 2 17 8 13  }
    8-> { 3 15 9 4 14 10 0 16 5 2 12 11 1 17 7  }
    9-> { 0 8 13 14 3 10 7 11 6 16 1  }
    10-> { 14 13 3 5 8 12 11 9 1 15 7 6 16 17  }
    11-> { 17 6 0 5 10 15 13 9 8 12 16 1 2 14  }
    12-> { 15 13 2 10 16 0 8 7 1 11 6 14  }
    13-> { 0 12 9 16 10 2 3 14 11 4 15 1 17 7  }
    14-> { 16 10 17 1 9 8 5 0 13 15 3 7 4 11 12  }
    15-> { 7 8 12 2 1 0 11 14 10 13 16 6 17  }
    16-> { 14 0 13 12 8 4 5 3 15 2 11 10 9 17  }
    17-> { 11 14 6 2 1 5 8 15 0 7 10 13 16  }
Looking for cliques of size: 7........

Brute Force Algorithm:
----------------------------------------
0 1 5 8 11 14 17 , 0 1 8 11 12 14 15 , 0 1 8 11 14 15 17 , 0 1 11 12 13 14 15 , 0 1 11 13 14
    15 17 , 0 5 8 11 14 16 17 , 0 8 11 12 14 15 16 , 0 8 11 14 15 16 17 , 0 11 12 13 14 15 16
    , 0 11 13 14 15 16 17 ,

Greedy heuristic algorithm:
---------------------------------------
No clique of size 7

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 10 vertices
and has 43 edges.
The adjacency matrix for the generated random graph is:
0 1 1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 0
1 1 0 1 1 1 1 1 1 1
0 1 1 0 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 0 1 1
1 1 1 1 1 1 1 1 0 1
1 0 1 1 1 1 1 1 1 0
-----------------------
    0-> { 4 7 5 9 8 6 1 2  }
    1-> { 4 3 0 2 7 6 8 5  }
    2-> { 6 5 7 8 3 9 4 0 1  }
    3-> { 8 1 9 2 4 7 5 6  }
    4-> { 0 1 9 7 8 3 2 6 5  }
    5-> { 8 2 0 6 7 3 9 4 1  }
    6-> { 2 5 9 8 0 4 7 3 1  }
    7-> { 0 9 2 4 5 3 6 8 1  }
    8-> { 5 3 9 0 6 2 4 7 1  }
    9-> { 7 4 8 0 6 3 2 5  }
Looking for cliques of size: 8........

Brute Force Algorithm:
----------------------------------------
0 1 2 4 5 6 7 8 , 0 2 4 5 6 7 8 9 ,

Greedy heuristic algorithm:
----------------------------------------
0 1 2 4 5 6 7 8
Yes, there is a clique of size 8

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 18 vertices
and has 43 edges.
The adjacency matrix for the generated random graph is:
0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 1
0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 0
1 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1 0
1 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0
1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1
0 1 1 0 1 0 0 0 1 1 1 0 0 1 1 0 0 0
0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0
0 0 1 1 0 0 0 1 0 1 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0
1 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 1
1 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0
0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0
0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 1
1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0
----------------------
    0-> { 5 6 17 12 4 14  }
    1-> { 12 6 14 5 7  }
    2-> { 12 7 10  }
    3-> { 16 5 10 15  }
    4-> { 13 8 16 0 7  }
    5-> { 0 3 8 6 9 1  }
    6-> { 0 5 9 1 17 15  }
    7-> { 9 10 2 14 13 8 4 1  }
    8-> { 5 4 12 7  }
    9-> { 7 11 5 6 10  }
    10-> { 11 16 7 2 3 9  }
    11-> { 9 10 14  }
    12-> { 2 8 1 0  }
    13-> { 14 4 7 17  }
    14-> { 13 11 7 1 0  }
    15-> { 16 3 6  }
    16-> { 3 10 17 15 4  }
    17-> { 0 16 6 13  }
Looking for cliques of size: 11........

Brute Force Algorithm:
----------------------------------------


Greedy heuristic algorithm:
----------------------------------------
No clique of size 11

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 20 vertices
and has 110 edges.
The adjacency matrix for the generated random graph is:
0 1 1 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 0
1 0 1 1 1 1 0 0 1 0 0 1 1 0 1 0 1 0 1 1
1 1 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 0 1
0 1 1 0 1 1 0 1 1 0 0 1 1 1 1 1 0 1 0 1
1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 1
1 1 0 1 1 0 1 1 0 0 1 0 1 1 0 1 1 0 0 1
1 0 0 0 1 1 0 1 1 1 0 1 0 0 1 1 0 1 1 0
1 0 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0
0 1 1 1 0 0 1 1 0 0 0 0 1 1 0 0 0 1 1 1
1 0 0 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0
0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 1 1 1 1 1
0 1 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 0 1
1 1 1 1 0 1 0 1 1 1 1 1 0 0 0 1 0 1 0 0
1 0 0 1 0 1 0 1 1 1 0 0 0 0 0 1 0 1 1 1
1 1 0 1 0 0 1 1 0 1 1 0 0 0 0 0 0 1 1 0
1 0 1 1 0 1 1 1 0 0 1 1 1 1 0 0 0 1 0 1
1 1 1 0 1 1 0 1 0 1 1 1 0 0 0 0 0 1 0 0
1 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 0 0 0
1 1 0 0 0 0 1 1 1 1 1 0 0 1 1 0 0 0 0 1
0 1 1 1 1 1 0 0 1 0 1 1 0 1 0 1 0 0 1 0
-----------------------
    0-> { 6 13 12 18 4 9 14 17 2 7 1 16 5 15  }
    1-> { 3 2 4 5 16 0 19 8 12 11 14 18  }
    2-> { 4 12 15 8 1 0 19 11 3 16  }
    3-> { 19 8 1 12 4 13 17 14 15 7 2 5 11  }
    4-> { 2 9 5 0 1 3 19 16 6  }
    5-> { 4 16 19 15 1 10 12 7 6 0 13 3  }
    6-> { 0 9 14 17 18 7 8 4 11 5 15  }
    7-> { 9 8 11 14 15 6 12 0 5 16 13 3 18  }
    8-> { 3 7 2 18 17 13 6 1 19 12  }
    9-> { 12 7 6 14 4 0 17 10 16 18 13  }
    10-> { 18 9 5 16 17 14 19 15 12  }
    11-> { 15 19 7 17 2 6 12 16 1 3  }
    12-> { 9 2 0 3 5 7 11 15 17 1 10 8  }
    13-> { 0 3 17 18 8 15 19 7 9 5  }
    14-> { 9 6 0 7 18 3 10 17 1  }
    15-> { 11 2 5 7 19 13 3 12 17 10 0 6  }
    16-> { 5 10 17 4 1 9 7 0 11 2  }
    17-> { 6 9 8 11 0 13 16 3 10 12 14 15  }
    18-> { 19 0 8 6 10 13 14 9 7 1  }
    19-> { 3 11 18 5 2 4 15 13 1 10 8  }
Looking for cliques of size: 6........

Brute Force Algorithm:
----------------------------------------
0 6 7 9 14 18 ,

Greedy heuristic algorithm:
----------------------------------------
No clique of size 6

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 9 vertices
and has 36 edges.
The adjacency matrix for the generated random graph is:
0 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1
1 1 0 1 1 1 1 1
1 1 1 0 1 1 1 1
1 1 1 1 0 1 1 1
1 1 1 1 1 0 1 1
1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 0
----------------------
    0-> { 2 3 4 6 5 1 7 8  }
    1-> { 5 2 6 7 0 4 3 8  }
    2-> { 3 0 1 4 6 5 8 7  }
    3-> { 2 0 6 8 4 7 1 5  }
    4-> { 0 2 5 6 8 3 1 7  }
    5-> { 1 0 8 2 4 7 6 3  }
    6-> { 7 1 0 2 3 4 5 8  }
    7-> { 6 8 1 5 0 3 2 4  }
    8-> { 3 7 5 4 0 6 1 2  }
Looking for cliques of size: 2........

Brute Force Algorithm:
----------------------------------------
0 1 , 0 2 , 0 3 , 0 4 , 0 5 , 0 6 , 0 7 , 0 8 ,

Greedy heuristic algorithm:
----------------------------------------
Yes, there is a clique of size 2

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 20 vertices
and has 110 edges.
The adjacency matrix for the generated random graph is:
0 0 1 0 1 0 1 1 1 0 0 0 0 1 1 1 0 0 1 1
0 0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0
1 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 0 0 1 1
0 0 0 0 0 0 1 0 0 1 0 1 0 1 1 0 1 1 0 1
1 1 1 0 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 1
0 1 1 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1 0 0
1 1 0 1 1 1 0 0 0 1 0 1 0 0 1 1 1 0 0 1
1 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 0 1 0
1 1 1 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0 1 1
0 1 1 1 1 0 1 1 0 0 1 1 0 1 0 0 0 0 0 1
0 0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 0 1 0 0
0 1 1 1 0 1 1 0 0 1 0 0 1 0 1 1 1 1 1 1
0 1 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1
1 1 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 1 0
1 1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 0 0
1 1 1 0 1 0 1 0 0 0 0 1 1 1 1 0 1 1 1 0
0 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 1 0
0 1 0 1 1 1 0 0 0 0 1 1 1 0 0 1 0 0 1 1
1 1 1 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1 0 1
1 0 1 1 1 0 1 0 1 1 0 1 1 0 0 0 0 1 1 0
----------------------
    0-> { 8 6 18 7 13 4 19 14 2 15  }
    1-> { 4 18 11 5 13 12 9 7 8 15 17 14 16 2 6  }
    2-> { 15 19 18 9 10 11 5 4 0 12 7 1 8  }
    3-> { 11 17 13 19 9 16 6 14  }
    4-> { 7 13 17 1 6 0 9 2 16 10 15 18 19  }
    5-> { 17 8 6 1 14 2 11 10  }
    6-> { 0 5 16 11 9 4 14 15 19 3 1  }
    7-> { 4 8 14 10 16 0 1 18 9 12 2  }
    8-> { 7 0 18 5 1 16 19 14 13 2  }
    9-> { 13 2 6 1 11 4 7 3 19 10  }
    10-> { 7 13 2 17 9 4 14 12 5  }
    11-> { 3 15 1 18 19 6 14 9 16 12 17 2 5  }
    12-> { 15 1 17 11 7 2 19 16 10  }
    13-> { 4 9 18 1 0 3 10 14 15 8 16  }
    14-> { 7 15 16 5 11 13 1 0 6 8 10 3  }
    15-> { 2 14 11 12 18 1 16 13 6 0 4 17  }
    16-> { 6 7 14 8 11 1 18 4 3 15 12 13  }
    17-> { 5 4 3 12 18 1 11 10 19 15  }
    18-> { 8 0 2 1 11 13 15 7 17 16 19 4  }
    19-> { 2 11 3 0 9 8 18 12 17 6 4  }
Looking for cliques of size: 3........

Brute Force Algorithm:
----------------------------------------
0 2 4 , 0 2 7 , 0 2 8 , 0 2 15 , 0 2 18 , 0 2 19 , 0 4 6 , 0 4 7 , 0 4 13 , 0 4 15 , 0 4 18 , 0 4 19 , 0 6 14 ,
    0 6 15 , 0 6 19 , 0 7 8 , 0 7 14 , 0 7 18 , 0 8 13 , 0 8 14 , 0 8 18 , 0 8 19 , 0 13 14 , 0 13 15 , 0 13 18
    , 0 14 15 , 0 15 18 , 0 18 19 ,

Greedy heuristic algorithm:
----------------------------------------
0 2 4
Yes, there is a clique of size 3

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 16 vertices
and has 93 edges.
The adjacency matrix for the generated random graph is:
0 1 0 1 1 0 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 1 0 1 1 1 1 1 1 1 0 1 0 1 0 1
1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 1
1 1 1 0 0 1 0 1 0 0 1 0 0 0 1 1
0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 1 0 1 1 1 0 0 0 1 1 0
1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 1
1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 0
1 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1
1 1 1 1 0 1 0 1 1 1 1 0 1 0 1 1
1 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1
1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1
1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 0
1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 0
----------------------
    0-> { 12 7 14 4 13 11 1 8 3 15 10 9 6  }
    1-> { 4 15 6 7 3 2 0 13 10 11 9 8 5 14 12  }
    2-> { 8 11 6 1 4 3 15 9 7 5 13  }
    3-> { 12 1 15 2 11 8 0 7 14 9 10  }
    4-> { 1 14 0 15 5 2 7 10  }
    5-> { 9 14 4 6 12 8 1 11 10 7 2 15 13  }
    6-> { 1 2 7 14 8 5 9 13 0  }
    7-> { 9 0 1 6 15 10 12 8 3 5 2 4 11  }
    8-> { 2 6 11 9 14 15 13 0 3 5 7 1 12 10  }
    9-> { 7 5 12 8 11 6 2 1 0 3 13  }
    10-> { 15 14 7 12 1 0 5 4 11 3 8  }
    11-> { 2 12 8 0 9 3 1 14 5 15 10 7  }
    12-> { 0 3 11 15 9 5 7 10 8 13 14 1  }
    13-> { 15 0 1 8 6 14 2 12 5 9  }
    14-> { 4 0 6 5 8 10 13 11 1 3 12  }
    15-> { 1 10 13 12 4 7 3 8 0 2 5 11  }
Looking for cliques of size: 9........

Brute Force Algorithm:
----------------------------------------
0 1 3 7 8 10 11 12 15 ,

Greedy heuristic algorithm:
----------------------------------------
Yes, there is a clique of size 9

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 13 vertices
and has 9 edges.
The adjacency matrix for the generated random graph is:
0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 1 0 0 1 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 0 0 0 0 0
----------------------
    0-> { 7  }
    1-> { 5  }
    2-> { 11 3  }
    3-> { 12 2  }
    4-> { Isolated Vertex! }
    5-> { 1  }
    6-> { 12  }
    7-> { 0 8 9  }
    8-> { 10 7  }
    9-> { 7  }
    10-> { 8  }
    11-> { 2  }
    12-> { 3 6  }
Looking for cliques of size: 2........

Brute Force Algorithm:
----------------------------------------
0 7 ,

Greedy heuristic algorithm:
----------------------------------------
0 7
Yes, there is a clique of size 2
```

```
Random graph generation:
The graph has 12 vertices
and has 45 edges.
The adjacency matrix for the generated random graph is:
0 1 0 0 1 1 1 1 1 0 1 1
1 0 1 0 0 1 1 1 0 1 0 0
0 1 0 1 0 1 1 1 0 0 1 1
0 0 1 0 1 1 1 1 0 0 1 1
1 0 0 1 0 1 0 0 1 0 1 0
1 1 1 1 1 0 0 1 1 1 1 1
1 1 1 1 0 0 0 1 1 0 1 1
1 1 1 1 0 1 1 0 1 1 1 1
1 0 0 0 1 1 1 1 0 1 0 0
0 1 0 0 0 1 0 1 1 0 1 1
1 0 1 1 1 1 1 1 0 1 0 1
1 0 1 1 0 1 1 1 0 1 1 0
----------------------
    0-> { 7 8 4 6 5 10 1 11  }
    1-> { 9 7 0 2 5 6  }
    2-> { 10 5 7 6 11 1 3  }
    3-> { 6 5 11 7 10 4 2  }
    4-> { 0 5 10 8 3  }
    5-> { 2 8 7 9 10 11 3 4 0 1  }
    6-> { 3 10 7 0 2 11 8 1  }
    7-> { 10 6 0 5 11 9 1 2 3 8  }
    8-> { 5 0 4 7 9 6  }
    9-> { 11 1 7 5 8 10  }
    10-> { 7 6 2 5 11 4 3 0 9  }
    11-> { 9 7 5 10 3 6 2 0  }
Looking for cliques of size: 4........

Brute Force Algorithm:
----------------------------------------
0 1 5 7 , 0 1 6 7 , 0 4 5 8 , 0 4 5 10 , 0 5 7 8 , 0 5 7 10 , 0 5 7 11 , 0 5 10 11 , 0 6 7 8 , 0 6 7 10 , 0 6 7 11 , 0 6 10 11 , 0 7 10 11 ,

Greedy heuristic algorithm:
----------------------------------------
0 1 5 7
Yes, there is a clique of size 4

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 17 vertices
and has 61 edges.
The adjacency matrix for the generated random graph is:
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1
1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 0 0
0 1 0 1 1 0 1 1 0 1 1 0 0 0 1 0 0
0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 0
0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 1 0
0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 1 1
0 0 1 1 1 0 0 0 1 1 1 1 1 0 1 0 0
1 1 1 1 0 1 0 0 0 0 0 1 0 0 0 1 1
0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 1 1
0 1 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1
0 0 1 1 1 1 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 1 1 0 1 0 0 1 1 0 1 0
0 1 0 1 0 0 1 0 0 0 0 1 0 1 1 1 1
1 1 0 0 0 0 0 0 0 0 1 1 1 0 1 1 0
0 1 1 0 0 1 1 0 1 0 0 0 1 1 0 1 0
0 0 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
1 0 0 0 0 1 0 1 1 1 0 0 1 0 0 0 0
-----------------------
    0-> { 13 1 16 7  }
    1-> { 14 7 0 4 2 8 9 13 12  }
    2-> { 14 6 7 3 10 1 9 4  }
    3-> { 7 15 6 2 10 12 5  }
    4-> { 10 6 15 8 1 11 2  }
    5-> { 14 7 10 15 3 16  }
    6-> { 14 2 11 4 3 12 8 9 10  }
    7-> { 2 3 5 16 1 11 15 0  }
    8-> { 16 15 14 4 6 1  }
    9-> { 15 11 6 16 1 2  }
    10-> { 4 5 13 2 3 6  }
    11-> { 6 13 12 9 7 4 15  }
    12-> { 13 14 6 11 16 3 1 15  }
    13-> { 15 14 12 0 11 10 1  }
    14-> { 2 1 13 5 6 8 15 12  }
    15-> { 13 8 9 14 3 4 7 5 11 12  }
    16-> { 8 7 12 0 9 5  }
Looking for cliques of size: 3........

Brute Force Algorithm:
----------------------------------------
0 1 7 , 0 1 13 , 0 7 16 ,

Greedy heuristic algorithm:
----------------------------------------
0 1 7
Yes, there is a clique of size 3

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 17 vertices
and has 71 edges.
The adjacency matrix for the generated random graph is:
0 0 0 0 0 1 1 1 1 1 0 1 1 1 1 0 0
0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0
0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 1
0 1 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1
0 0 0 0 0 0 1 1 1 1 0 1 1 0 0 0 1
1 1 0 1 0 0 0 1 1 0 1 1 0 1 1 0 1
1 0 0 0 1 0 0 1 0 1 0 1 1 0 0 0 1
1 0 1 1 1 1 1 0 1 1 0 0 0 1 1 1 0
1 0 1 0 1 1 0 1 0 1 1 0 1 1 1 0 1
1 0 0 0 1 0 1 1 1 0 1 0 1 0 0 1 0
0 0 1 1 0 1 0 0 1 1 0 0 0 1 1 1 0
1 0 1 0 1 1 1 0 0 0 0 0 1 0 0 0 1
1 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0
1 1 0 1 0 1 0 1 1 0 1 0 1 0 0 1 1
1 1 1 0 0 1 0 1 1 0 1 0 0 0 0 1 0
0 1 1 1 0 0 0 1 0 1 1 0 1 1 1 0 0
0 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0
----------------------
    0-> { 7 8 6 14 9 5 13 12 11  }
    1-> { 5 14 3 12 15 13  }
    2-> { 14 10 16 7 15 11 8  }
    3-> { 5 16 13 1 7 15 10  }
    4-> { 12 9 8 7 6 11 16  }
    5-> { 14 1 0 3 8 10 11 13 7 16  }
    6-> { 0 12 16 11 7 4 9  }
    7-> { 0 14 9 6 4 5 13 2 3 8 15  }
    8-> { 0 10 13 9 12 4 16 5 14 7 2  }
    9-> { 4 0 8 7 12 6 15 10  }
    10-> { 8 2 14 5 13 15 3 9  }
    11-> { 6 5 16 0 12 2 4  }
    12-> { 6 4 15 8 9 13 1 0 11  }
    13-> { 8 16 3 5 7 0 12 15 10 1  }
    14-> { 2 5 0 15 7 10 1 8  }
    15-> { 14 12 2 13 7 3 1 10 9  }
    16-> { 6 2 8 3 13 11 4 5  }
Looking for cliques of size: 5........

Brute Force Algorithm:
----------------------------------------
0 5 7 8 13 , 0 5 7 8 14 ,

Greedy heuristic algorithm:
----------------------------------------
0 5 7 8 13
Yes, there is a clique of size 5

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 10 vertices
and has 26 edges.
The adjacency matrix for the generated random graph is:
0 0 1 1 1 1 0 0 0 1
0 0 1 0 1 0 0 1 0 0
1 1 0 0 0 1 1 1 1 1
1 0 0 0 1 0 0 1 1 1
1 1 0 1 0 1 0 1 1 1
1 0 1 0 1 0 1 0 1 1
0 0 1 0 0 1 0 0 0 0
0 1 1 1 1 0 0 0 0 1
0 0 1 1 1 1 0 0 0 1
1 0 1 1 1 1 0 1 1 0
----------------------
    0-> { 3 2 5 9 4  }
    1-> { 4 7 2  }
    2-> { 9 6 0 5 7 1 8  }
    3-> { 0 4 7 8 9  }
    4-> { 8 3 1 7 5 0 9  }
    5-> { 2 4 6 8 0 9  }
    6-> { 2 5  }
    7-> { 3 4 1 2 9  }
    8-> { 4 3 5 2 9  }
    9-> { 2 8 0 5 7 3 4  }
Looking for cliques of size: 4........

Brute Force Algorithm:
----------------------------------------
0 2 5 9 , 0 3 4 9 , 0 4 5 9 ,

Greedy heuristic algorithm:
----------------------------------------
0 2 5 9
Yes, there is a clique of size 4

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 11 vertices
and has 34 edges.
The adjacency matrix for the generated random graph is:
0 1 0 0 1 0 1 1 1 0 1
1 0 0 0 0 0 1 1 1 1 0
0 0 0 0 1 1 1 0 0 1 1
0 0 0 0 1 1 0 0 0 1 1
1 0 1 1 0 1 1 0 0 1 1
0 0 1 1 1 0 1 0 0 1 1
1 1 1 0 1 1 0 1 1 1 1
1 1 0 0 0 0 1 0 0 0 1
1 1 0 0 0 0 1 0 0 1 1
0 1 1 1 1 1 1 0 1 0 1
1 0 1 1 1 1 1 1 1 1 0
-----------------------
    0-> { 7 4 8 10 6 1  }
    1-> { 7 8 9 6 0  }
    2-> { 10 4 6 9 5  }
    3-> { 10 4 5 9  }
    4-> { 5 10 0 2 3 6 9  }
    5-> { 4 9 10 3 6 2  }
    6-> { 7 2 4 10 8 1 5 0 9  }
    7-> { 0 6 1 10  }
    8-> { 9 0 10 1 6  }
    9-> { 8 10 5 1 2 3 6 4  }
    10-> { 3 2 9 4 0 8 6 5 7  }
Looking for cliques of size: 4........

Brute Force Algorithm:
----------------------------------------
0 1 6 7 , 0 1 6 8 , 0 4 6 10 , 0 6 7 10 , 0 6 8 10 ,

Greedy heuristic algorithm:
----------------------------------------
0 1 6 7
Yes, there is a clique of size 4

Program ended with exit code: 0
```

```
The graph has 10 vertices
and has 38 edges.
The adjacency matrix for the generated random graph is:
0 0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 0 0 1
1 1 1 0 1 1 0 1 1 1
1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 0 1 0 1 1
1 1 1 0 1 1 0 1 1 0
1 1 0 1 1 0 1 0 1 1
1 1 0 1 1 1 1 1 0 0
1 1 1 1 1 1 0 1 0 0
----------------------
    0-> { 6 2 8 4 3 9 5 7  }
    1-> { 4 8 6 9 3 5 7 2  }
    2-> { 0 3 4 6 5 1 9  }
    3-> { 2 4 9 1 5 0 8 7  }
    4-> { 8 1 3 5 2 9 7 0 6  }
    5-> { 6 8 4 2 1 9 3 0  }
    6-> { 8 1 0 5 2 7 4  }
    7-> { 9 8 4 6 1 0 3  }
    8-> { 4 6 1 5 7 0 3  }
    9-> { 7 1 3 4 5 0 2  }
Looking for cliques of size: 4........

Brute Force Algorithm:
----------------------------------------
0 2 3 4 , 0 2 3 5 , 0 2 3 9 , 0 2 4 5 , 0 2 4 6 , 0 2 4 9 , 0 2 5 6 , 0 2 5 9 , 0 3 4 5 , 0 3 4
    7 , 0 3 4 8 , 0 3 4 9 , 0 3 5 8 , 0 3 5 9 , 0 3 7 8 , 0 3 7 9 , 0 4 5 6 , 0 4 5 8 , 0 4 5 9
    , 0 4 6 7 , 0 4 6 8 , 0 4 7 8 , 0 4 7 9 , 0 5 6 8 , 0 6 7 8 ,

Greedy heuristic algorithm:
----------------------------------------
0 2 3 4
Yes, there is a clique of size 4

Program ended with exit code: 0
```

```
Random graph generation:
The graph has 10 vertices
and has 15 edges.
The adjacency matrix for the generated random graph is:
0 0 0 1 0 1 0 1 0 1
0 0 0 0 1 1 0 0 0 0
0 0 0 1 1 0 0 0 1 1
1 0 1 0 1 0 0 0 1 0
0 1 1 1 0 0 1 0 0 0
1 1 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0 1
0 0 1 1 0 0 0 0 0 0
1 0 1 0 0 1 0 1 0 0
-----------------------
    0-> { 3 7 5 9  }
    1-> { 4 5  }
    2-> { 8 4 3 9  }
    3-> { 8 0 4 2  }
    4-> { 6 1 3 2  }
    5-> { 1 9 0  }
    6-> { 4  }
    7-> { 0 9  }
    8-> { 3 2  }
    9-> { 7 5 2 0  }
Looking for cliques of size: 8........

Brute Force Algorithm:
----------------------------------------


Greedy heuristic algorithm:
----------------------------------------
No clique of size 8

Program ended with exit code: 0
```

Appendix 3: (Complete code used for analysis)

```
#include <iostream>
#include <cmath>
#include "randgen.h"
#include <vector>
using namespace std;

const int MAX = 100;

// Stores the vertices
int store[MAX], n;
// Graph
int graph[MAX][MAX];
// Degree of the vertices
int d[MAX];

// Function to check if the given set of vertices
// in store array is a clique or not
bool is_clique(int b)
{
    // Run a loop for all the set of edges
    // for the select vertex
    for (int i = 0; i < b; i++) {
        for (int j = i + 1; j < b; j++)
            // If any edge is missing
            if (graph[store[i]][store[j]] == 0)
                return false;
    }
    return true;
}

// Function to print the clique
void print(int n)
{
    for (int i = 0; i < n; i++)
        cout << store[i] << " ";
    cout << ", ";
}

// Function to find all the cliques of size s
```

```
void findCliques(int i, int l, int s)
{
   // Check if any vertices from i+1 can be inserted
   for (int j = i; j <= n - (s - l); j++)

      // If the degree of the graph is sufficient
      if (d[j] >= s - 1) {
         // Add the vertex to store
         store[l] = j;

         // If the graph is not a clique of size k
         // then it cannot be a clique
         // by adding another edge
         if (is_clique(l + 1)){
            // If the length of the clique is
            // still less than the desired size
            if (l + 1 < s)
               // Recursion to add vertices
               findCliques(j, l + 1, s);
            // Size is met
            else
               print(l + 1);
         }
      }
}
void GenRandomGraphs(int NOEdge, int NOVertex)
{
   RandGen gen;
   int i, j, edge[NOEdge][2], count;

   // Initialize the adjacency matrix to false
   for (i = 0; i < NOVertex; i++) {
      for (j = 0; j < NOVertex; j++) {
         graph[i][j] = 0;
         d[i] = 0;
      }
   }
   i = 0;
   // Assign random values to the number
   // of vertex and edges of the graph,
```

```
// Using rand().
while (i < NOEdge) {
    edge[i][0] = gen.RandInt(1,INT_MAX) % NOVertex + 1;
    edge[i][1] = gen.RandInt(1,INT_MAX) % NOVertex + 1;
    // Check if edge already exists
    if (edge[i][0] == edge[i][1]) {
        continue;
    } else if (graph[edge[i][0]-1][edge[i][1]-1] || graph[edge[i][1]-1][edge[i][0]-1]) {
        continue;
    } else {
        // Add edge to adjacency matrix
        graph[edge[i][0]-1][edge[i][1]-1] = true;
        graph[edge[i][1]-1][edge[i][0]-1] = true;
        d[edge[i][0]-1] += 1; // increment degree of first vertex of the edge
        d[edge[i][1]-1] += 1; // increment degree of first vertex of the edge
        i++;
    }
}
// Print the adjacency matrix
cout << "The adjacency matrix for the generated random graph is:" << endl;
for (i = 0; i < NOVertex; i++) {
    for (j = 0; j < NOVertex; j++) {
        cout << graph[i][j] << " ";
    }
    cout << endl;
}
cout << "----------------------" << endl;
for (i = 0; i < NOVertex; i++) {
        count = 0;
        cout << "\t" << i<< "-> { ";
        for (j = 0; j < NOEdge; j++) {
            if (edge[j][0] == i + 1) {
                cout << edge[j][1]-1 << " ";
                count++;
            }
            else if (edge[j][1] == i + 1) {
                cout << edge[j][0]-1 << " ";
                count++;
            }
            else if (j == NOEdge - 1 && count == 0)
```

```
                // Print “Isolated vertex”
                // for the vertex having
                // no degree.
                cout << "Isolated Vertex!";
            }
            cout << " }" << endl;
        }
}
bool greedy_clique(int k) {

   if (k == 1) {
      return true;
   }
   vector<int> clique;
   vector<int> vertices;
   for (int i = 0; i < n; ++i) {
      vertices.push_back(i);
   }

   for (int i = 0; i < vertices.size(); ++i) {
      clique.clear();
      clique.push_back(vertices[i]);
      for (const auto& v : vertices) {
         if (find(clique.begin(), clique.end(), v) != clique.end()) {
            continue;
         }
         bool isNext = true;
         for (const auto& u : clique) {
            if (!graph[u][v]) {
               isNext = false;
               break;
            }
         }
         if (isNext) {
            clique.push_back(v);
            if (k <= clique.size()) {
               for(int x=0; x < clique.size(); x++){
                  cout << clique[x] << " ";
               }
               cout << endl;
```

```
                return true;
            }
        }
    }
  }
  return false;
}
// Driver code
int main()
{

  int e;
  RandGen gen;
  cout << "Random graph generation: " << endl;
  n = gen.RandInt(7,20); //random between 7 and 20 simply
  cout << "The graph has " << n << " vertices" << endl;

  e = gen.RandInt(0,((n * (n - 1)) / 2));

  cout << "and has " << e << " edges." << endl;

  // Function call
  GenRandomGraphs(e, n);

  int k = gen.RandInt(2,8); //random between 2 and 15 simply
  cout << "Looking for cliques of size: " << k << "......." << endl;

  cout << endl << "Brute Force Algorithm: " << endl;
  cout << "---------------------------------------" << endl;
  findCliques(0, 0, k);
  cout << endl;

  cout << endl << "Greedy heuristic algorithm: " << endl;
  cout << "---------------------------------------" << endl;

  if(greedy_clique(k)){
     cout << "Yes, there is a clique of size " << k << endl;
  }
  else{
     cout << "No clique of size " << k << endl;
```

```
    }
    cout << endl;
    return 0;
}
```