CS301 – Term Project

Project Report

Group Members:

TUGAY GARİB

FURKAN REHA TUTAŞ

DERYA BENSU ÇAKAR

MELTEM ARSLAN

EMRE HİLMİ SONGUR

Semester: 2019 Fall

Instructor: Husnu Yenigun

Faculty of Engineering and Natural Sciences

Sabancı Üniversitesi

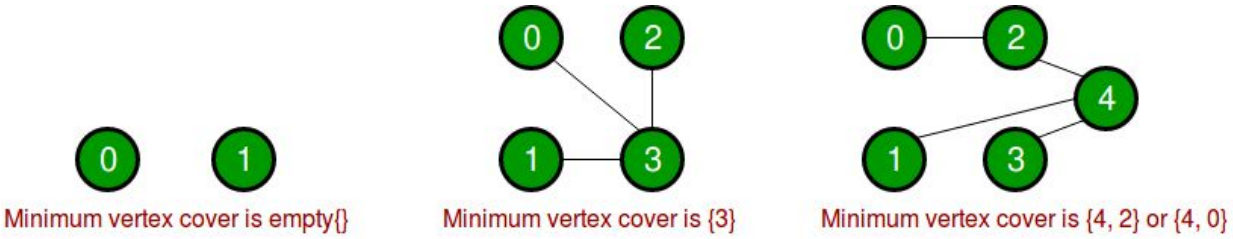
1) PROBLEM DESCRIPTION

A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.

More formally,

A n-node undirected graph G(V, E) with node set V and edge set E; a positive integer k with $k \leq n$. Is there a subset W of V having size at most k and such that for every edge {u, v} in E at least one of u and v belongs to W?

Basic Examples of Vertex Cover:



This problem can be used in practice. For example let's assume that one needs to put security cameras at every intersection at a city to record vehicles plate numbers if they violate the traffic rules. In this instance, roads will be represented by edges and intersections will be represented by the vertices. It is best to use as few cameras as possible to decrease the cost of installation and maintenance work.

Vertex Cover is Np Complete.

Proof:

Show that

A) Vertex Cover is in NP.

B) An NP Complete Problem can be reduced to Vertex Cover in polynomial time.

A) Vertex Cover is in NP

If any problem is in NP, then, given a guess (a solution) to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the guess in polynomial time.

The guess for the vertex cover problem is a subset V'of V, which contains the vertices in the vertex cover. We can check whether the set V' is a vertex cover of size k' which is smaller than k using the following algorithm (for a graph G(V, E))

```
verifyGuess (G (V, E), V'){
k' = 0;
foreach vertex v'in  V' {
        remove all edges adjacent to v' from E;
        k' ++;
        }
if E == empty && k' <= k {
```

```
        print("Solution is correct");
}
else {
        print("Solution is wrong");
        }
}
```

In the worst case, V' will be equal to V and every v' will be adjacent to each v in V. Therefore, in the worst case, the algorithm will have $O(V^2)$ which is polynomial time.

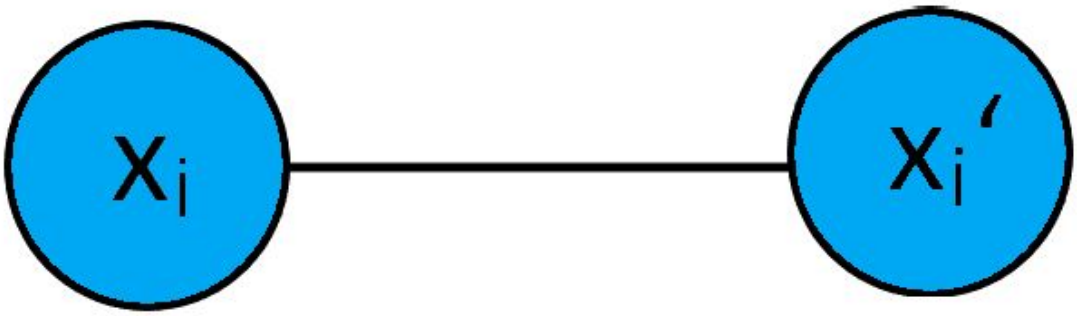
As a conclusion, Vertex cover is in NP.

B) The NP Complete Problem can be reduced to Vertex Cover in polynomial time

In order to prove Vertex Cover is NP Hard, a reduction algorithm from 3-SAT (Known NP-Complete Problem) to Vertex Cover in polynomial time is provided.

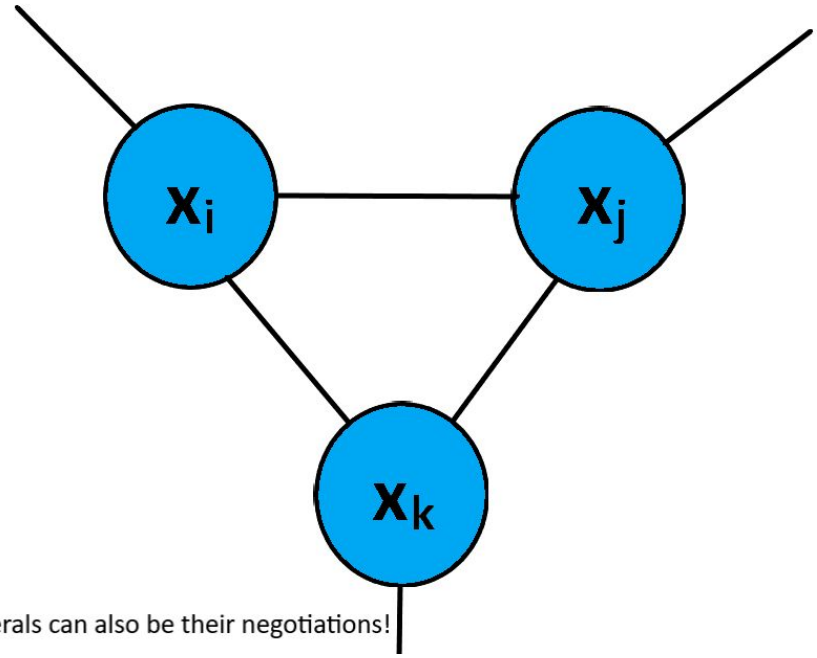
Steps:

1. Take an instance of 3-SAT (in form of ..... AND (x_i V x_j V x_k) AND…..)
2. Force each literal to be either TRUE or FALSE by creating a pair of nodes for each literal and its negation. The following is an example illustration for a literal x_i in 3-SAT.



3. Create a "gadget" for each conjunct in 3-SAT instances (x_i V x_j V x_k).

   Create 3 vertex triangles where each vertex is connected to other 2 vertices and also connected to the corresponding literal which is constructed in step 2.



Note that: Literals can also be their negotiations!

4. Choose an appropriate K value for the instance of vertex cover. K is chosen as L + 2M where L is the number of literals in 3-SAT instance and M is the number of conjuncts in 3-SAT instance or number of "gadgets" in Vertex Cover instance.

5. Proof of choice of K as L + 2M

   Assume that there is a satisfying assignment of literals in 3-SAT instance. Make TRUE literals part of the Vertex Cover instance in corresponding graph G (V, E). L nodes are covered. By choosing one vertex for each literal, each edge between literal vertices (x_i and x_i') is covered. Since, the 3-SAT instance is satisfied, at least one literal in each conjunct must be chosen as TRUE indicating that in each triangle, there is at least 1 vertex whose outgoing edge (the edge going outside of the triangle to its corresponding literal which is marked as TRUE) is covered. As a result, at most there are 2 more

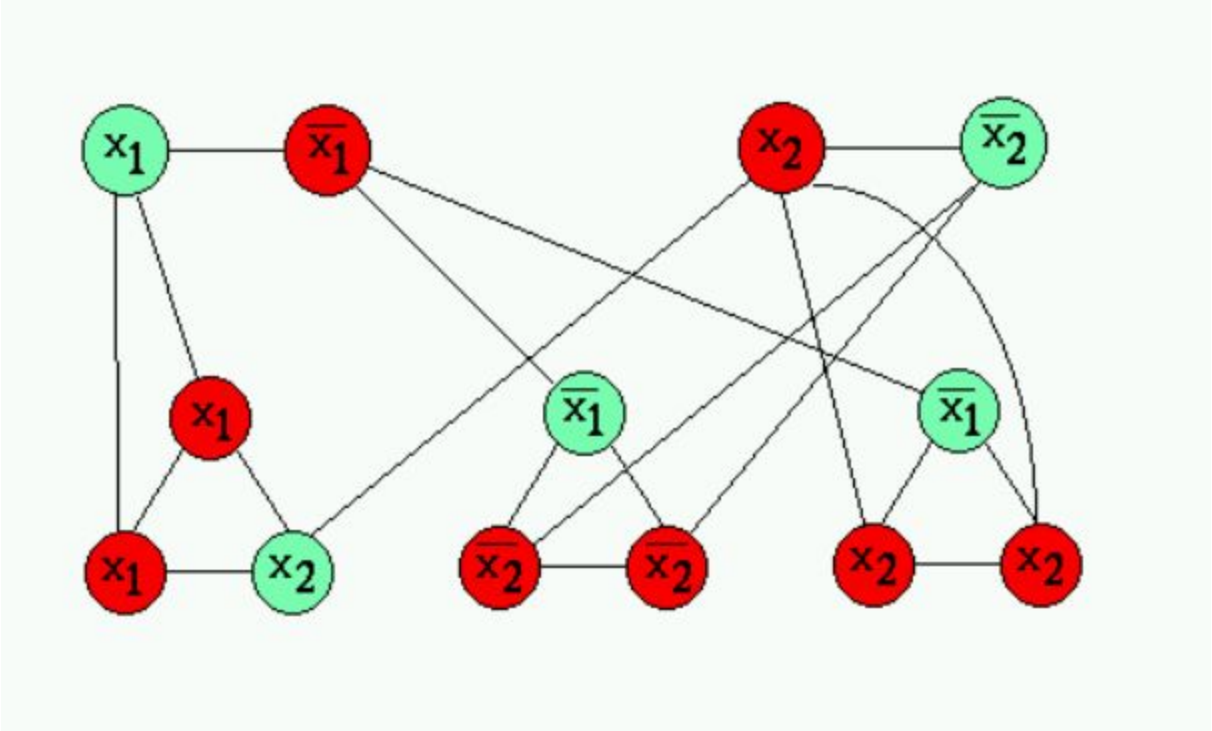
vertices to be covered per each triangle. Therefore, at most 2 x M (number of triangles or conjuncts) vertices need to be covered. In total, maximum L + 2M vertices are covered in Vertex Cover instance.

Assume that there is a Vertex Cover instance with L + 2M vertices or less. In order to cover each edge between literal pairs (x_i and x_i'), it is guaranteed that at least one of the vertices (x_i or x_i') is covered since the edge between them cannot be covered in any other way. The other 2M (or less) vertices need to cover triangles. In order for 2M vertices to be enough for covering all triangles, at least one of the outgoing edges from each triangle must be covered by vertices from literal pairs (x_i or x_i'). Since covered vertices in literal pairs (x_i or x_i') are marked as TRUE, each triangle (corresponding conjunct in 3-SAT instance) has at least one vertex (corresponding literal in corresponding conjunct) which is marked as TRUE. Then this has to be a satisfying assignment of literals in the corresponding 3-SAT instance.

6. Finally, the input size is M conjuncts with L literals and output size is exactly a graph consisting of 3M + 2L vertices. This can be constructed in polynomial time with an iterative algorithm.

Example Execution of Reduction Algorithm for an 3-SAT instance

(x1 V x1 V x2) (!x1 V !x2 V !x2) (!x1 V x2 V x2) with answer x1 = FALSE and x2 = TRUE:



As a conclusion:

Since Vertex Cover is shown to be in NP class and at least as hard as 3-SAT problem which is NP-Complete, Vertex Cover Problem is also an NP-Complete problem.

2) ALGORITHM DESCRIPTION

No algorithm has been found such that it can solve Vertex Cover in polynomial time. However, there are some heuristic algorithms such that they solve Vertex Cover in polynomial

time without guaranteeing an optimal solution. In this project, the greedy algorithm approach to Vertex Cover analyzed.

The Greedy Algorithm Steps:

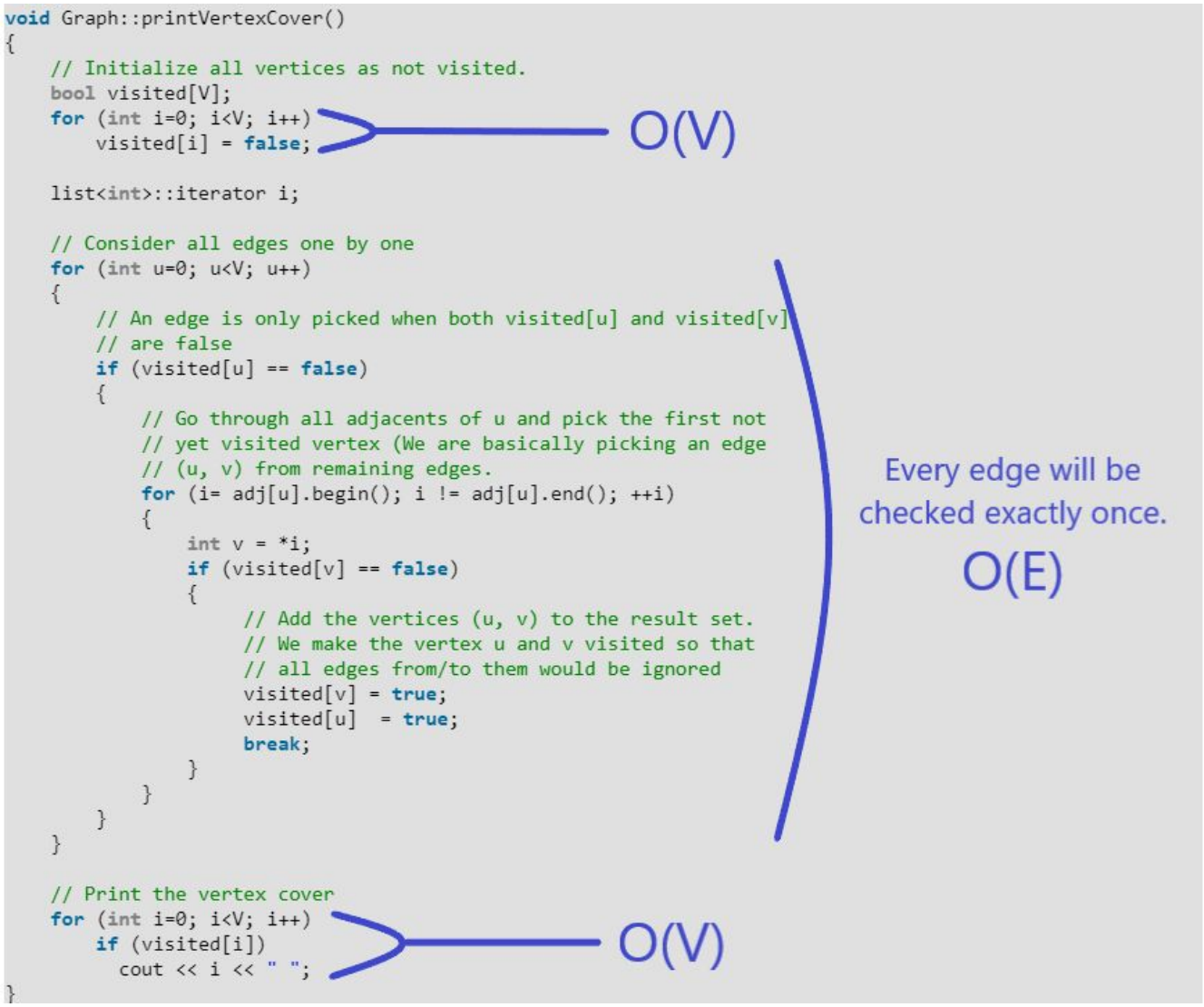
1. Create an empty set called V'.
2. Call the set of edges in the graph as UC.
3. Iterate over the next edge called (u,v) from UC.
4. If chosen edge is not visited (both u and v are not visited) insert (u,v) to V' and proceed, otherwise go back to Step 3
5. Mark u and v as visited, if there are still edges to iterate go back to Step 3, otherwise proceed.
6. Print visited vertices as Vertex Cover of the graph.

The following algorithm works in the same manner as described above.

```
void Graph::printVertexCover()
{
    // Initialize all vertices as not visited.
    bool visited[V];
    for (int i=0; i<V; i++)
        visited[i] = false;

    list<int>::iterator i;

    // Consider all edges one by one
    for (int u=0; u<V; u++)
    {
        // An edge is only picked when both visited[u] and visited[v]
        // are false
        if (visited[u] == false)
        {
            // Go through all adjacents of u and pick the first not
            // yet visited vertex (We are basically picking an edge
            // (u, v) from remaining edges.
            for (i= adj[u].begin(); i != adj[u].end(); ++i)
            {
                int v = *i;
                if (visited[v] == false)
                {
                     // Add the vertices (u, v) to the result set.
                     // We make the vertex u and v visited so that
                     // all edges from/to them would be ignored
                     visited[v] = true;
                     visited[u]  = true;
                     break;
                }
            }
        }
    }

    // Print the vertex cover
    for (int i=0; i<V; i++)
        if (visited[i])
          cout << i << " ";
}
```

3) ALGORITHM ANALYSIS

Worst case asymptotic running time of the algorithm as follows:



O(V) + O(E) + O(V) = O(V + E)

Running time of printVertexCover is O(V + E)

Additionally, this greedy algorithm has a ratio bound as follows:

Steps:

1. At any time of vertex cover selection let A be the set of selected edges and C be the set of selected vertices.
2. Note that no 2 edges in A share endpoints. Therefore, 2*size(A) = Size(C)
3. In order to cover the edges in A, any vertex cover (also the optimum one) has to include at least one endpoint.
4. Therefore size(A) ≤ size(Optimum Cover) → size(C) ≤ 2 x size(Optimum Cover)
5. Ratio Bound = maximumSize(Heuristic Solution) / minimumSize(Optimum Cover) = 2

* Note that size represents the number of vertices in the corresponding Vertex Cover solution.

As a result, regardless of input size (V and E), the vertex cover set of the greedy algorithm will not be larger than two times the optimal vertex cover.

4) EXPERIMENTAL ANALYSIS

a) Running Time Experimental Analysis

In order to experimentally analyse the algorithm running time complexity, the following functions are used to calculate standard deviation, standard error, sample mean and confidence level intervals.

1) Standard Deviation

```
float calculateSD(vector<float> & data) {
    float sum = 0.0, mean, stadardDeviation = 0.0;
    for (int i = 0; i < data.size(); i++) {
        sum += data[i];
    }
    mean = sum / data.size();
    for (int j = 0; j < data.size(); j++) {
        stadardDeviation += pow(data[j] - mean, 2);
    }
    stadardDeviation = sqrt(stadardDeviation / data.size());
    return stadardDeviation;
}
```

The code given in the above figure is the implementation of standard deviation formula.

$$\sigma = \sqrt{\frac{\sum(X-\mu)^2}{n}}$$

where,

σ = population standard deviation
$\sum$ = sum of...
μ = population mean
n = number of scores in sample.

2) Standard Error

```
float calculateStandardError(float standardDeviation, int N) {
    return standardDeviation / sqrt(N);
}
```

The code given in the above figure is the implementation of standard error formula.

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

3) Running Time Algorithm

```
void getRunningTime(vector<float> & runningTimes) {
    float totalTime = 0.0;
    int N = runningTimes.size();
    for (int i = 0; i < N; i++) {
        totalTime += runningTimes[i];
    }

    float standardDeviation = calculateSD(runningTimes);


    float m = totalTime / N;

    const float tval90 = 1.645;

    const float tval95 = 1.96;

    float sm = calculateStandardError(standardDeviation, N);

    float upperMean90 = m + tval90 * sm;
    float lowerMean90 = m - tval90 * sm;

    float upperMean95 = m + tval95 * sm;
    float lowerMean95 = m - tval95 * sm;

    cout << "Mean Time: " << m << " ms" << endl;
    cout << "SD: " << standardDeviation << endl;
    cout << "Standard Error: " << sm << endl;
    cout << "%90-CL: " << upperMean90 << " - " << lowerMean90 << endl;
    cout << "%95-CL: " << upperMean95 << " - " << lowerMean95 << endl;
    runningTimes.clear();
}
```

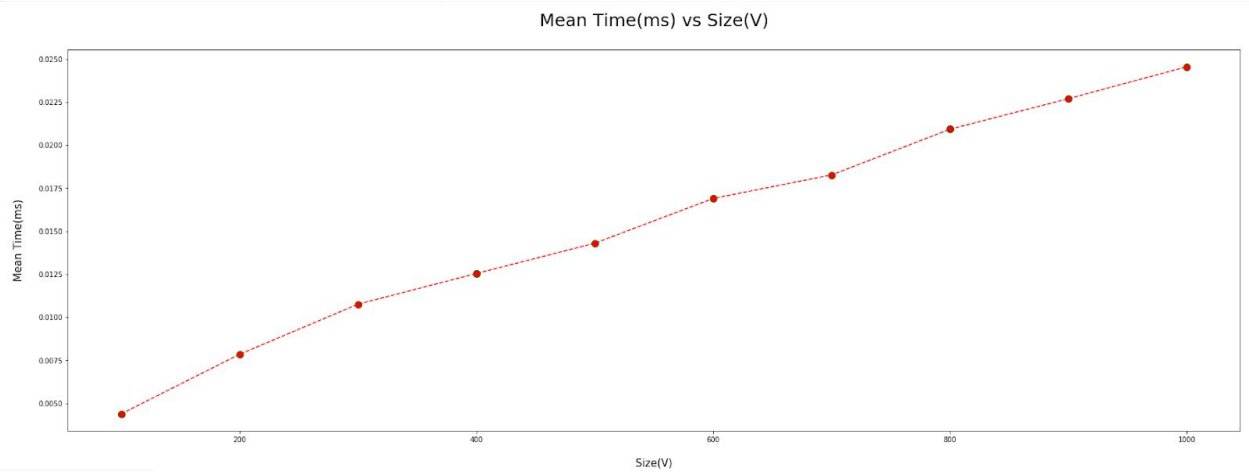
For the sake of simplicity, assume t-value for confidence level %90 is 1.645 and t-value for %95 confidence level is 1.96. The code given in the above figure calculates, mean time in milliseconds, standard deviation, standard error, upper and lower limits for both confidence levels %90 and %95 for the corresponding running time iterations.

Note that our heuristic approach has O(V + E) time complexity, therefore one of inputs will be kept constant while changing the other one to observe the relationship between inputs and time complexity.

Keep Number of Edges Constant (E = 200)

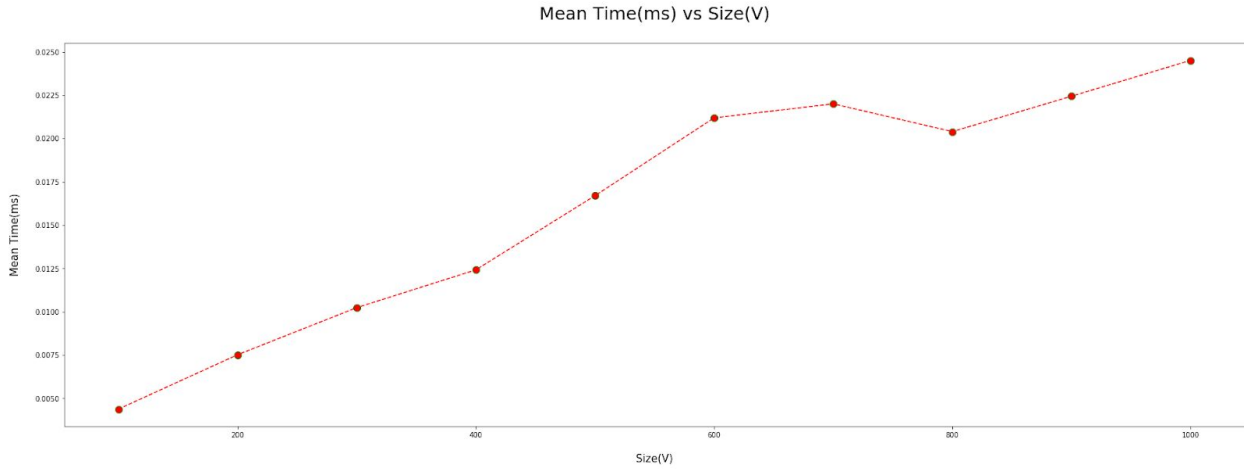
100 Number of Iterations per each Input Size

Size(V)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
100	0.004384	0.000515	0.000052	0.004299 - 0.004469	0.004283 - 0.004485
200	0.007847	0.000755	0.000075	0.007723 - 0.007971	0.007699 - 0.007995
300	0.01076	0.002847	0.000285	0.010292 - 0.011228	0.010202 - 0.011318
400	0.012544	0.002435	0.000244	0.012143 - 0.012945	0.012067 - 0.013021
500	0.014306	0.001152	0.000115	0.014117 - 0.014495	0.014080 - 0.014532
600	0.016906	0.004602	0.00046	0.016149 - 0.017663	0.016004 - 0.017808
700	0.018275	0.001248	0.000125	0.018070 - 0.018480	0.018030 - 0.018520
800	0.020931	0.004988	0.000499	0.020111 - 0.021751	0.019953 - 0.021909
900	0.022707	0.004177	0.000418	0.022020 - 0.023394	0.021888 - 0.023526
1000	0.024564	0.002197	0.00022	0.024203 - 0.024925	0.024133 - 0.024995



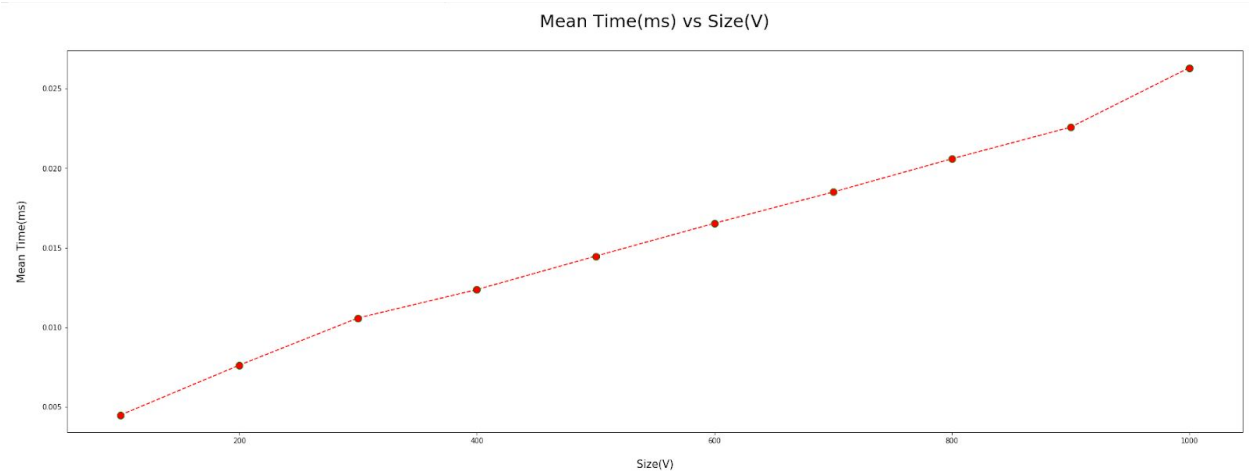
1000 Number of Iterations per each Input Size

Size(V)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
100	0.004375	0.00117	0.000037	0.004314 - 0.004436	0.004303 - 0.004448
200	0.007513	0.001253	0.00004	0.007448 - 0.007578	0.007436 - 0.007591
300	0.010233	0.001747	0.000055	0.010142 - 0.010324	0.010125 - 0.010341
400	0.012415	0.002408	0.000076	0.012289 - 0.012540	0.012265 - 0.012564
500	0.016695	0.006723	0.000213	0.016346 - 0.017045	0.016279 - 0.017112
600	0.021192	0.008523	0.00027	0.020748 - 0.021635	0.020663 - 0.021720
700	0.022	0.008065	0.000255	0.021580 - 0.022419	0.021500 - 0.022500
800	0.020399	0.002544	0.00008	0.020267 - 0.020532	0.020241 - 0.020557
900	0.022442	0.00253	0.00008	0.022310 - 0.022574	0.022285 - 0.022599
1000	0.024515	0.003396	0.000107	0.024339 - 0.024692	0.024305 - 0.024726



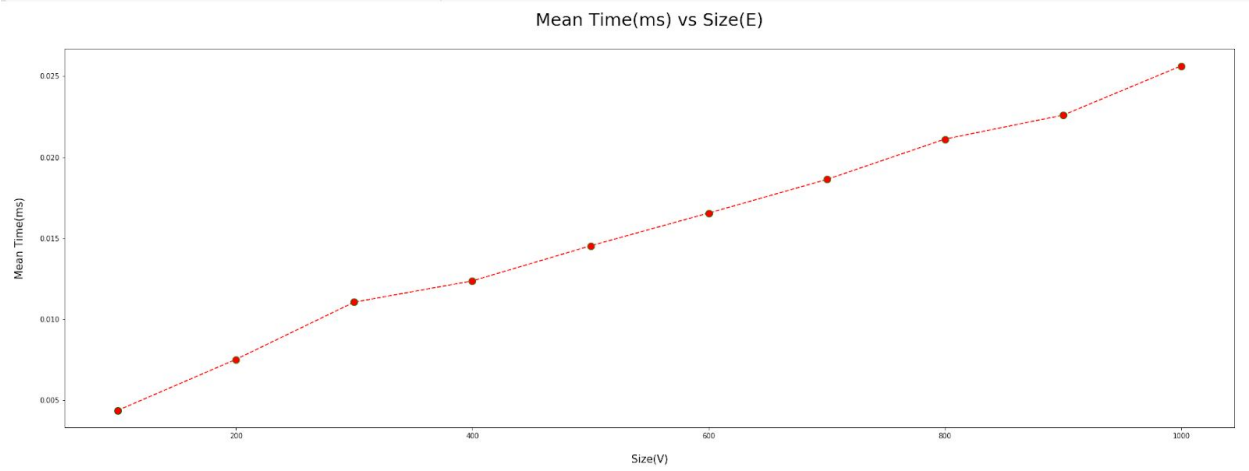
5000 Number of Iterations per each Input Size

Size(V)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
100	0.004496	0.001502	0.000021	0.004461 - 0.004531	0.004455 - 0.004538
200	0.007627	0.001637	0.000023	0.007589 - 0.007665	0.007582 - 0.007672
300	0.010591	0.002957	0.000042	0.010523 - 0.010660	0.010509 - 0.010673
400	0.012372	0.001932	0.000027	0.012327 - 0.012417	0.012319 - 0.012426
500	0.014473	0.002284	0.000032	0.014420 - 0.014526	0.014410 - 0.014536
600	0.016536	0.002602	0.000037	0.016475 - 0.016596	0.016464 - 0.016608
700	0.018498	0.00429	0.000061	0.018398 - 0.018598	0.018379 - 0.018617
800	0.020589	0.004418	0.000062	0.020486 - 0.020692	0.020467 - 0.020712
900	0.022571	0.003055	0.000043	0.022500 - 0.022642	0.022487 - 0.022656
1000	0.026308	0.007534	0.000107	0.026132 - 0.026483	0.026099 - 0.026517



10000 Number of Iterations per each Input Size

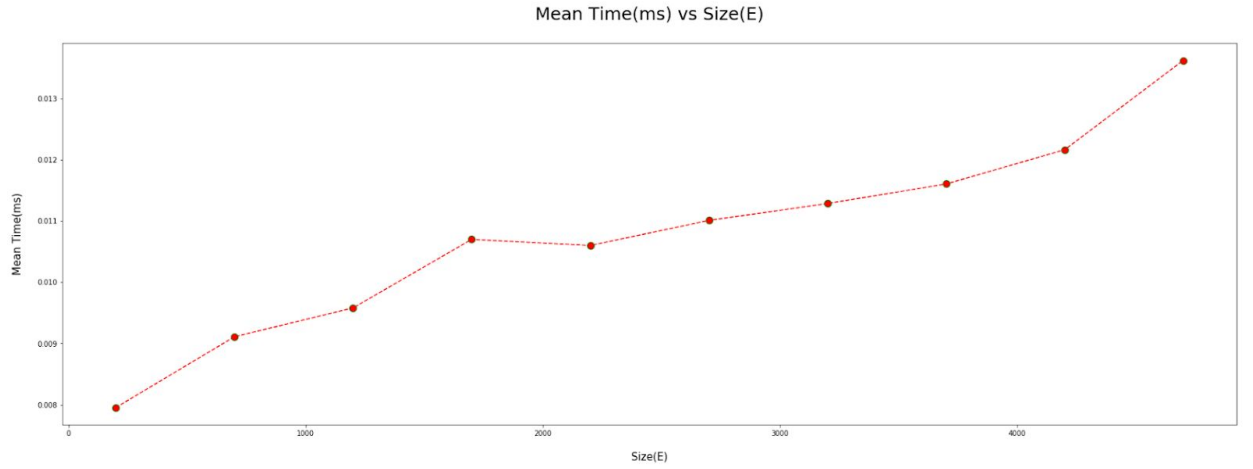
Size(V)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
100	0.004381	0.001179	0.000012	0.004362 - 0.004400	0.004358 - 0.004404
200	0.007527	0.001461	0.000015	0.007503 - 0.007551	0.007498 - 0.007555
300	0.011056	0.004305	0.000043	0.010985 - 0.011127	0.010972 - 0.011140
400	0.012367	0.001816	0.000018	0.012337 - 0.012397	0.012332 - 0.012403
500	0.014533	0.002346	0.000023	0.014495 - 0.014572	0.014487 - 0.014579
600	0.016554	0.003521	0.000035	0.016496 - 0.016612	0.016485 - 0.016623
700	0.018625	0.003798	0.000038	0.018562 - 0.018687	0.018550 - 0.018699
800	0.021117	0.004541	0.000045	0.021042 - 0.021191	0.021028 - 0.021206
900	0.022595	0.003435	0.000034	0.022538 - 0.022651	0.022527 - 0.022662
1000	0.025634	0.006642	0.000066	0.025524 - 0.025743	0.025503 - 0.025764



Keep Number of Vertices Constant (V = 200)

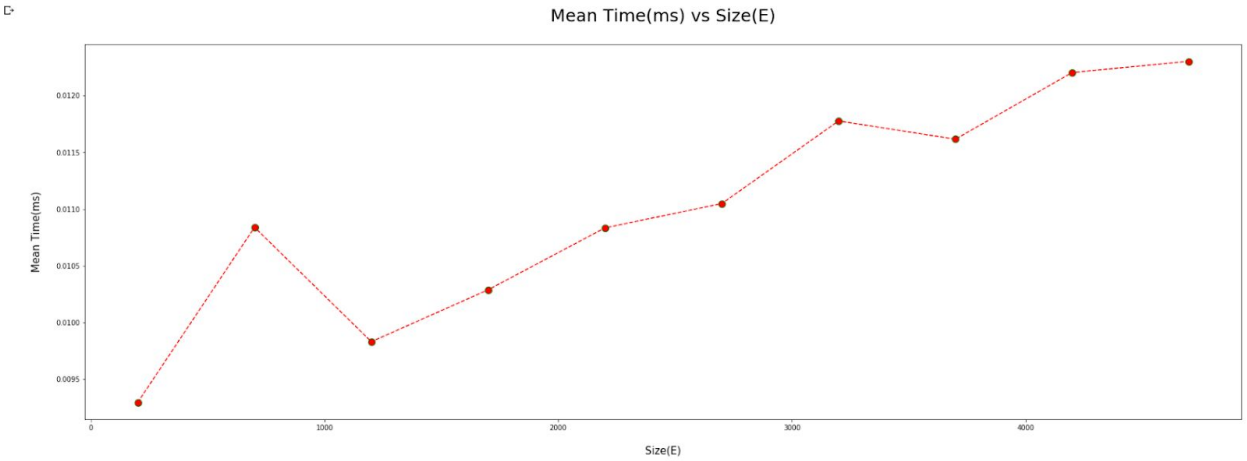
100 Number of Iterations per each Input Size

Size(E)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
200	0.007952	0.001645	0.000165	0.007681 - 0.008223	0.007630 - 0.008274
700	0.009109	0.002109	0.000211	0.008762 - 0.009456	0.008696 - 0.009522
1200	0.00958	0.001089	0.000109	0.009401 - 0.009759	0.009367 - 0.009793
1700	0.0107	0.003173	0.000317	0.010178 - 0.011222	0.010078 - 0.011322
2200	0.0106	0.002037	0.000204	0.010265 - 0.010935	0.010201 - 0.010999
2700	0.01101	0.00265	0.000265	0.010574 - 0.011446	0.010491 - 0.011529
3200	0.011286	0.002062	0.000206	0.010947 - 0.011625	0.010882 - 0.011690
3700	0.011605	0.00134	0.000134	0.011385 - 0.011825	0.011342 - 0.011868
4200	0.012165	0.002348	0.000235	0.011779 - 0.012551	0.011705 - 0.012625
4700	0.013625	0.003075	0.000307	0.013119 - 0.014131	0.013022 - 0.014228



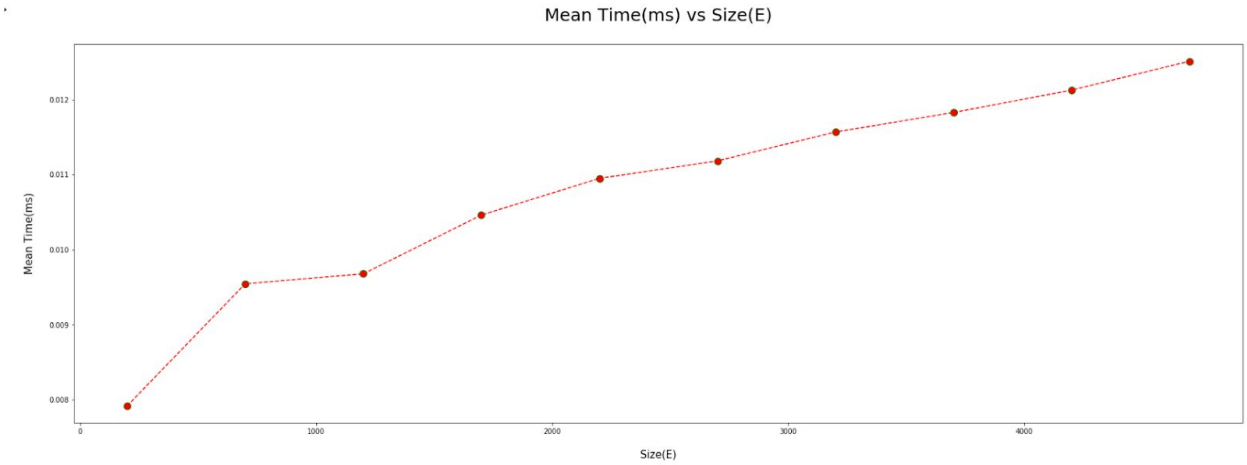
1000 Number of Iterations per each Input Size

Size(E)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
200	0.009295	0.004512	0.000143	0.009060 - 0.009529	0.009015 - 0.009574
700	0.010839	0.004803	0.000152	0.010590 - 0.011089	0.010542 - 0.011137
1200	0.00983	0.002045	0.000065	0.009724 - 0.009936	0.009703 - 0.009957
1700	0.010288	0.001978	0.000063	0.010185 - 0.010391	0.010165 - 0.010410
2200	0.010834	0.003037	0.000096	0.010676 - 0.010992	0.010646 - 0.011023
2700	0.011048	0.002292	0.000072	0.010929 - 0.011167	0.010906 - 0.011190
3200	0.011777	0.003077	0.000097	0.011617 - 0.011937	0.011586 - 0.011968
3700	0.011616	0.002082	0.000066	0.011507 - 0.011724	0.011487 - 0.011745
4200	0.012204	0.002858	0.00009	0.012055 - 0.012352	0.012027 - 0.012381
4700	0.012303	0.003063	0.000097	0.012143 - 0.012462	0.012113 - 0.012493



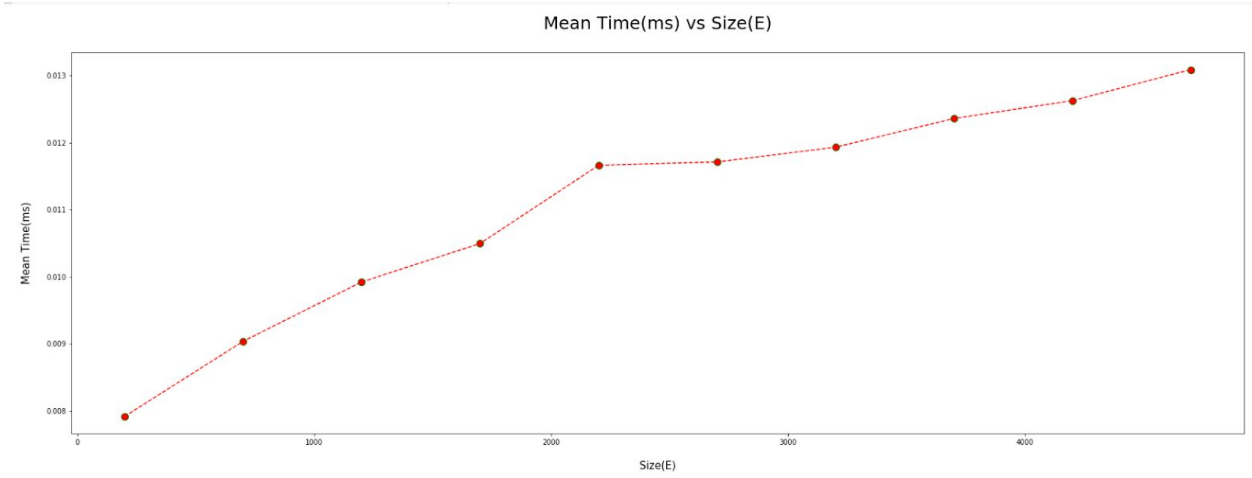
5000 Number of Iterations per each Input Size

Size(E)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
200	0.007919	0.001585	0.000022	0.007882 - 0.007956	0.007875 - 0.007963
700	0.009545	0.003052	0.000043	0.009474 - 0.009616	0.009460 - 0.009629
1200	0.009677	0.001593	0.000023	0.009640 - 0.009714	0.009633 - 0.009721
1700	0.01046	0.002577	0.000036	0.010400 - 0.010520	0.010389 - 0.010532
2200	0.010949	0.003427	0.000048	0.010869 - 0.011028	0.010854 - 0.011044
2700	0.011182	0.002469	0.000035	0.011125 - 0.011240	0.011114 - 0.011251
3200	0.011567	0.00293	0.000041	0.011499 - 0.011636	0.011486 - 0.011649
3700	0.011827	0.002566	0.000036	0.011768 - 0.011887	0.011756 - 0.011898
4200	0.012125	0.002643	0.000037	0.012064 - 0.012187	0.012052 - 0.012199
4700	0.012511	0.004833	0.000068	0.012398 - 0.012623	0.012377 - 0.012645



10000 Number of Iterations per each Input Size

Size(E)	Mean Time(ms)	Standard Deviation	Standard Error	% 90 - CL	% 95 - CL
200	0.007916	0.001777	0.000018	0.007886 - 0.007945	0.007881 - 0.007950
700	0.009036	0.001764	0.000018	0.009007 - 0.009065	0.009002 - 0.009071
1200	0.009921	0.003876	0.000039	0.009857 - 0.009984	0.009845 - 0.009997
1700	0.010495	0.002514	0.000025	0.010454 - 0.010536	0.010446 - 0.010544
2200	0.011661	0.011032	0.00011	0.011480 - 0.011843	0.011445 - 0.011877
2700	0.011712	0.006087	0.000061	0.011612 - 0.011812	0.011592 - 0.011831
3200	0.011931	0.004345	0.000043	0.011859 - 0.012002	0.011846 - 0.012016
3700	0.01236	0.009385	0.000094	0.012206 - 0.012515	0.012176 - 0.012544
4200	0.012626	0.007362	0.000074	0.012505 - 0.012747	0.012482 - 0.012771
4700	0.01309	0.006973	0.00007	0.012975 - 0.013204	0.012953 - 0.013226



b) Correctness

The heuristic algorithm for Vertex Cover problem doesn't always provide the minimum Vertex Cover for a given graph. However, it provides a Vertex Cover in all of the iterations since it traverses all the edges in the Graph.

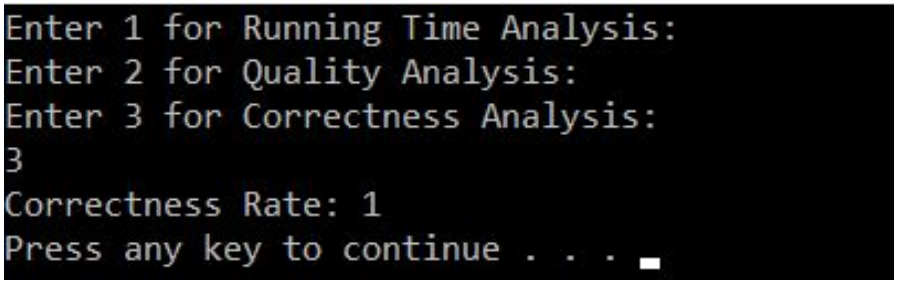
```
bool Graph::isVertexCover(vector<int> & vertexCover) {

    bool * visited = new bool[V];
    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }

    for (int k = 0; k < vertexCover.size(); k++) {
        list<int>::iterator i;
        visited[vertexCover[k]] = true;
        for (i = adj[vertexCover[k]].begin(); i != adj[vertexCover[k]].end(); ++i) {
            int v = *i;
            visited[v] = true;
        }
    }

    for (int i = 0; i < V; i++) {
        if (visited[i] == false) {
            return false;
        }
    }
    return true;
}
```

The algorithm in the Figure above, checks whether a set of vertices is a Vertex Cover or not, for a given Graph.

By utilizing the isVertexCover function the following code experimentally measures the correctness of the heuristic algorithm in terms of being a Vertex Cover or not.

```
else if (option == 3) {
    vector<int> vertexCover;
    double totalCorrect = 0;
    for (int i = 0; i < 10000; i++) {
        uniform_int_distribution<int> distribution1(1000, 10000);
        int vSize = distribution1(generator);
        uniform_int_distribution<int> distribution2(0, 20000);
        int eSize = distribution2(generator);

        Graph G(vSize);
        G.populate(eSize);
        G.returnVertexCover(vertexCover);
        if (G.isVertexCover(vertexCover)) {
            totalCorrect++;
        }

        vertexCover.clear();
    }
    cout << "Correctness Rate: " << totalCorrect / double(10000) << endl;
}
```

The heuristic algorithm is tested with 10000 Graphs with random number of vertices between [1000, 10000] and number of edges [Number of Vertices(will be explained later!), 20000]. The output is 1.



c) Ratio Bound

As proved in the Algorithm Analysis part, the heuristic algorithm has a ratio bound size(Vertex Cover) / size (Minimum Vertex Cover) ≤ 2. In the following part, this ratio bound will be checked by black box testing with relatively small graphs.

Let's define the quality of the heuristic algorithm as the following:

Quality = size(Minimum Vertex Cover) / size(Vertex Cover) = 1 / Ratio

Note that since quality measure is multiplicative inverse of Ratio, it must be a number in range [0.5, 1].

- Quality = 0.5 → Worst Case (Ratio Bound)
- Quality = 1 → Minimum Vertex Cover (Heuristic Solution Equals to Optimal Solution)

Brute Force Algorithm for Vertex Cover

```
int Graph::exactVertexCover() {
    vector<int> vertexCover;
    for (int i = 1; i <= V; i++) {
        vector<vector<int>> combinations = makeCombi(V, i);
        for (int k = 0; k < combinations.size(); k++) {
            for (int j = 0; j < combinations[k].size(); j++) {
                list<int>::iterator i;
                vertexCover.push_back(combinations[k][j]);
                for (i = adj[combinations[k][j] - 1].begin(); i != adj[combinations[k][j] - 1].end(); ++i) {
                    int v = *i;
                    if (find(vertexCover.begin(), vertexCover.end(), v) == vertexCover.end()) {
                        vertexCover.push_back(v);
                    }
                }
            }
            if (vertexCover.size() == V) {
                cout << "Vertex Cover: " << endl;
                for (int l = 0; l < combinations[k].size(); l++) {
                    cout << combinations[k][l] - 1 << endl;
                }
                return combinations[k].size();
            }
            vertexCover.clear();
        }

    }
    return -1;
}
```

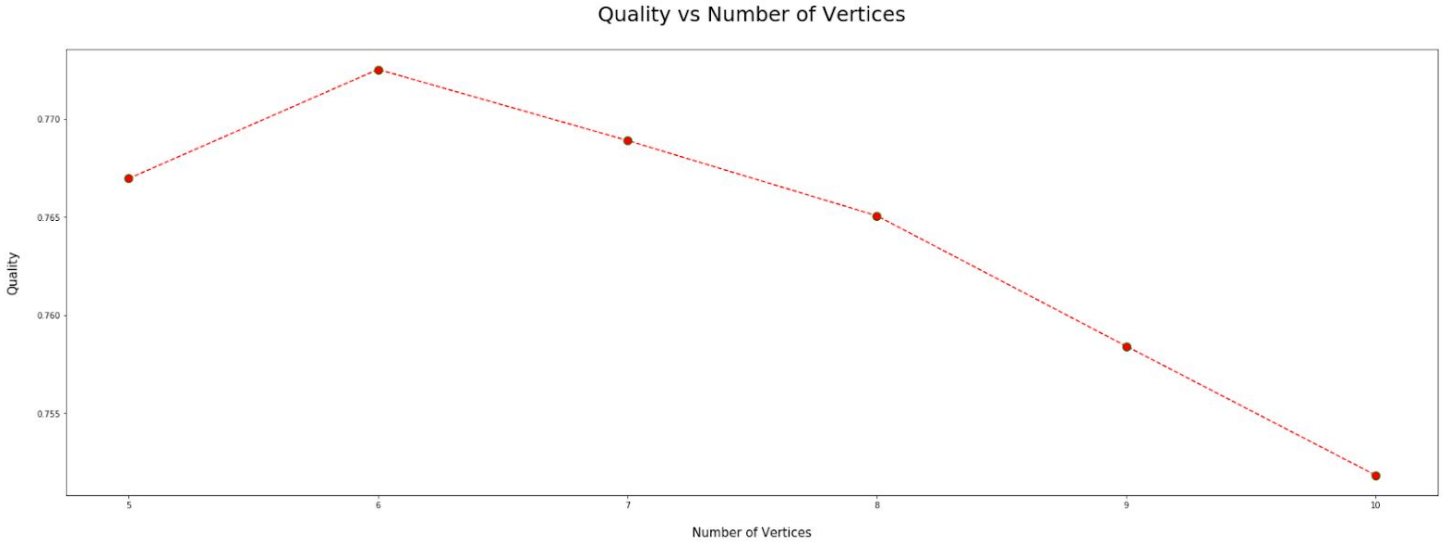
The given code in the above figure, simply iterates all possible combinations of vertices from smaller sizes to higher sizes until it finds a vertex cover. The algorithm iterates all vertices combinations which is in total 2^V - 1 in the worst case. The reason to develop the brute force algorithm is to check quality (1 / ratio bound) for relatively smaller graphs.

Quality = (1 / Ratio) where Ratio = size (Vertex Cover) / size (Min Vertex Cover)

Quality Vs Number of Vertices Table

# of Graphs	5 Vertices	6 Vertices	7 Vertices	8 Vertices	9 Vertices	10 Vertices
100	0.792917	0.81	0.78375	0.759833	0.737917	0.759167
1000	0.737958	0.767125	0.774542	0.764233	0.760242	0.759367
3000	0.76475	0.765267	0.769453	0.769453	0.761053	0.747522
5000	0.761133	0.76786	0.774032	0.759317	0.764552	0.766967
10000	0.766967	0.77253	0.768908	0.765071	0.758423	0.751849

As the figure above indicates, an increase in the number of vertices tends to lead to a decrease in the average quality of Vertex Cover of heuristic algorithms.

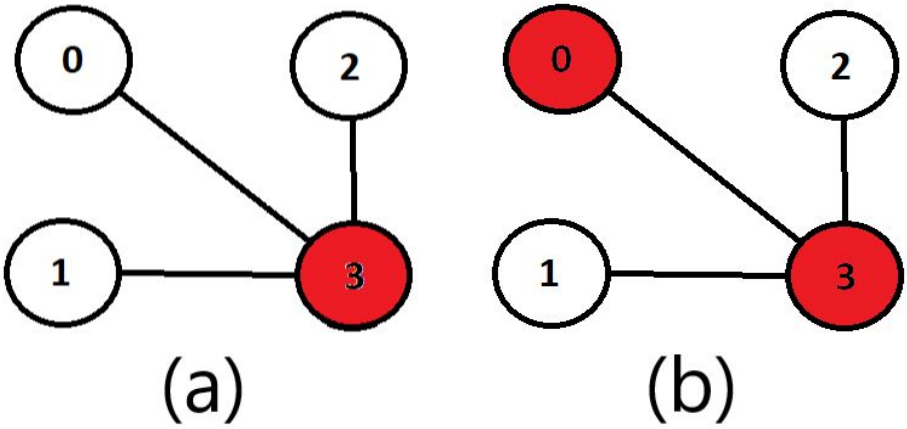


5) TESTING

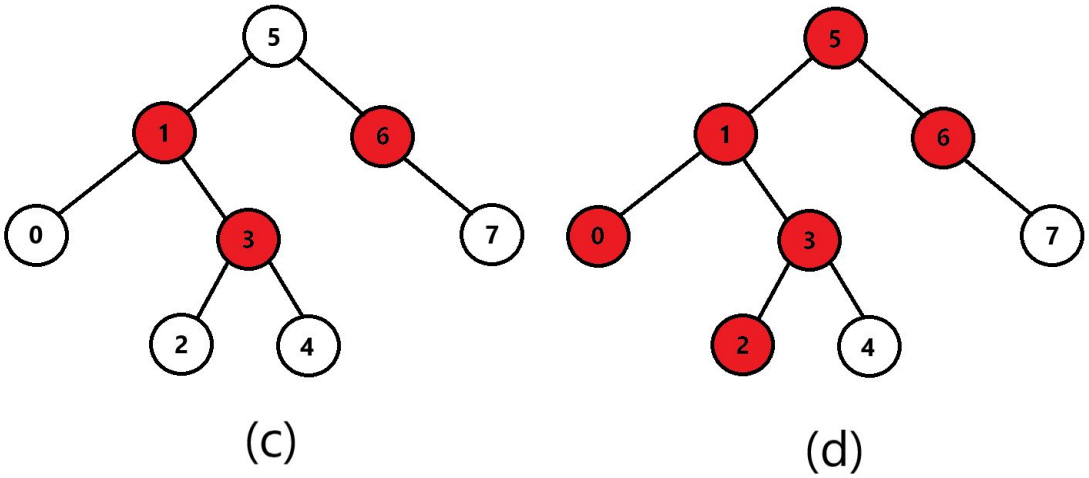
Blackbox testing is applied to test the heuristic algorithm. The following algorithm populates random graphs for testing and experimental analysis. Random graphs are created and checked for the algorithm's output for these graphs.

Note that the following algorithm (highlighted part) guarantees that every vertex in the graph has at least one edge. This part is added in order to make the algorithms more easy to develop such as checking whether a set of vertices is a minimum vertex cover or not, brute force for minimum vertex cover, etc.

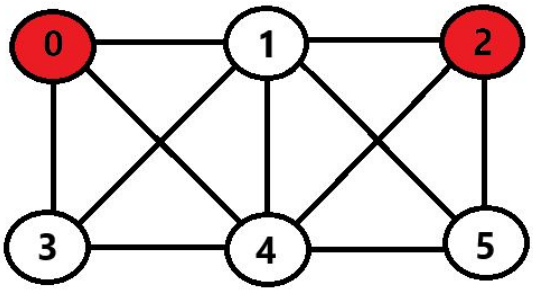
```
void Graph::populate(int E) {

    int fromV = 0;
    int toV = 0;
    uniform_int_distribution<int> distribution(0, V - 1);
    for (int i = 0; i < E; i++) {
        fromV = distribution(generator);
        toV = distribution(generator);

        while (fromV == toV || find(adj[toV].begin(), adj[toV].end(), fromV) != adj[toV].end()) {
            fromV = distribution(generator);
            toV = distribution(generator);
        }

        addEdge(fromV, toV);
    }

    list<int>::iterator i;
    for (int u = 0; u < V; u++)
    {
        if (adj[u].empty()) {
            toV = distribution(generator);
            addEdge(u, toV);
        }
    }
}
```



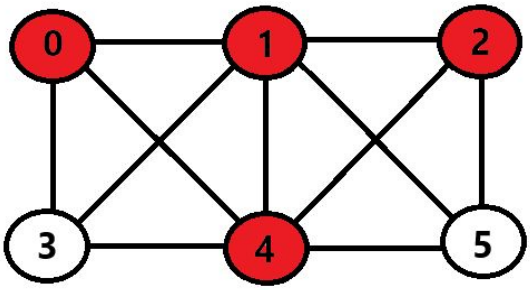
FAILED: Graph (a) represents the optimal vertex cover which only uses 1 vertex for this graph but our algorithm failed to find it for this instance. It generated the outcome given in (b) which needs 2 vertices.



FAILED: Graph (c) represents the optimal vertex cover which only uses 3 vertices for this graph but our algorithm failed to find it for this instance. It generated the outcome given in (d) which needs 6 vertices.

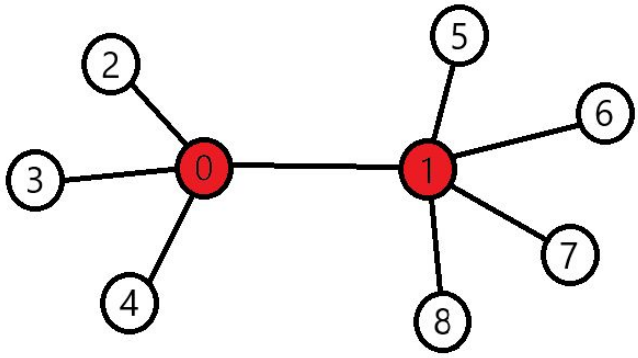


(e)



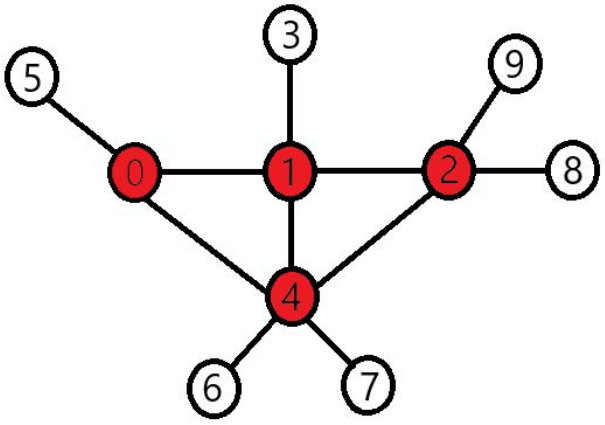
(f)

FAILED: Graph (e) represents the optimal vertex cover which only uses 2 vertices for this graph but our algorithm failed to find it for this instance. It generated the outcome given in (f) which needs 4 vertices.



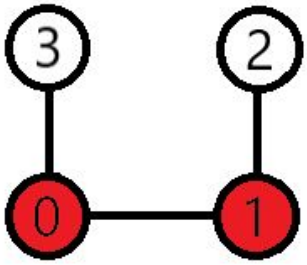
(g)

SUCCEDED: The algorithm produced the optimal result by only using 2 vertices.



(h)

SUCCEDED: The algorithm produced the optimal result by only using 4 vertices.



(i)

SUCCEDED: The algorithm produced the optimal result by only using 2 vertices.

6) DISCUSSION

As a conclusion, it is proven that Vertex Cover is a NP-Complete problem which is reduced from 3-SAT. There is no found algorithm that solves the problem in polynomial time. There are different heuristic algorithms that can solve the problem in linear time without guaranteeing the optimum solution. In this project, a greedy algorithm is analysed. The heuristic algorithm always provides a vertex cover but without guaranteeing minimum vertex cover. Additionally, it is proven that the heuristic algorithm has a ratio bound 2 which indicates that the cover of the heuristic algorithm will be at most 2 times bigger than the optimal solution for the corresponding instance.

In the experimental analysis of the report, firstly the running time of the heuristic algorithm O(V + E) is shown by keeping one input constant varying the other and vice versa. Secondly, correctness of the heuristic algorithm in terms of producing a vertex cover is shown by a polynomial time function isVertexCover with 10000 instances and varying number of vertices and edges. The heuristic algorithm has produced vertex cover for all the tested randomized graphs. Finally, a new metric quality of the heuristic algorithm is defined as 1 / Ratio where Ratio is size(vertex cover) / size(minimum vertex cover). In order to measure the quality of the heuristic algorithm, a brute force algorithm for minimum vertex cover for a small number of vertices is developed. The experiments indicate that the quality of the heuristic algorithm is around 0.76 and when the number of vertices increases, it tends to decrease with a small proportion.