FALL 2020

CS 301 - ALGORITHMS

GROUP PROJECT REPORT

MINIMUM DOMINATING SET PROBLEM

Instructor: Hüsnü Yenigün

Ayşenur Çerçi - 25024

Müzeyyen Alkap - 25046

Zeynep Akant - 25089

Ahmet Ölçüm - 24915

TABLE OF CONTENTS

Minimum Dominating Set Problem

Input: n-node undirected graph G(V,E)

Question: What is the smallest cardinality subset W of V such that for every node u in V-W (i.e. in V but not in W) there is a node w in W such that {u,w} is an edge of G?

1. Problem Description

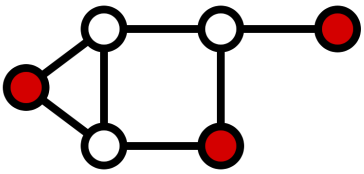
There exists a graph containing some amount of edges (represented as E) and vertices (represented as V). The dominating set of this graph consists of vertices from the original graph so that each vertex in the original graph is either a member of the dominating set or adjacent to at least one of the vertices in the dominating set. Asking what is the smallest cardinality of the dominating set that can be obtained from the original graph is an optimization problem.

In formal description, let G = (V, E) be a simple undirected graph. A dominating set in a graph G is a subset of vertices $S \subseteq V$ such that each vertex in V is either in S or is adjacent to some vertex in S. That is, for every vertex $u \in V-S$, there exists a vertex $v \in S$ such that $uv \in E$. A dominating set is minimal if S cannot be contracted further; that is, there exists no vertex $w \in S$ such that $S - \{w\}$ is also a dominating set in G. The problem of finding a minimal dominating set of minimum cardinality is a hard problem. (Iyer, K. Viswanathan, National Institute of Technology, Tiruchirappalli, CSE301 Lecture Notes)

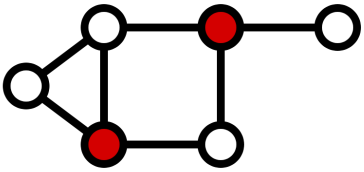
The applications of the Minimum Dominating Set Problem are quite rich. This model can be used in the study of social networks, design of wireless sensor networks, protein interaction

networks and covering codes. A real life example of the MDSP is such: Given a number of villages in a region, the goal is to locate radio stations with limited broadcasting range in some of these villages so that the messages can be broadcasted to all the villages in that region. Assuming that the fixed cost of locating a radio station is relatively high. In order to keep the total cost at minimum, we need to locate as few radio stations as possible, in other words, minimising the total number of the radio stations to be located.

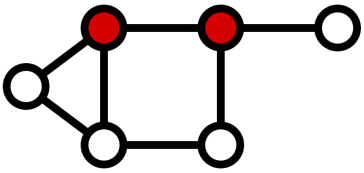
(a)



(b)



(c)



Here are the different solutions of the Minimum Dominating Set problem for the same graph. In figure (a), red vertices form a dominating set, however it does not have the minimum cardinality. On the other hand, in figure (b) and figure (c), red vertices form different dominating sets with the minimum cardinality 2.[1]

[1] https://en.wikipedia.org/wiki/Dominating_set

Theorem: MDSP is NP-Complete.

Proof: In order to prove that the Minimum Dominating Set problem is an NP-Complete problem, we need to show that this problem is both NP and NP-Hard.

The Minimum Dominating Set problem is an NP problem. That is, given a graph and an instance of a solution to the Minimum Dominating Set problem for this graph, say the subset V_1, we need to check every vertex in the graph is either in V_1, or adjacent to a vertex in V_1. It is obvious that these checks can be done within a polynomial time, and therefore it is an NP problem.

Verification:[2]

```
flag=true
for every vertex v in V:
  if v doesn't belong to Dominating Set:
     verify the set of edges
     corresponding to v
     if v is not adjacent
        to any of the vertices in DS,
        set flag=False and break
if (flag)
   solution is correct
else
   solution is incorrect
```

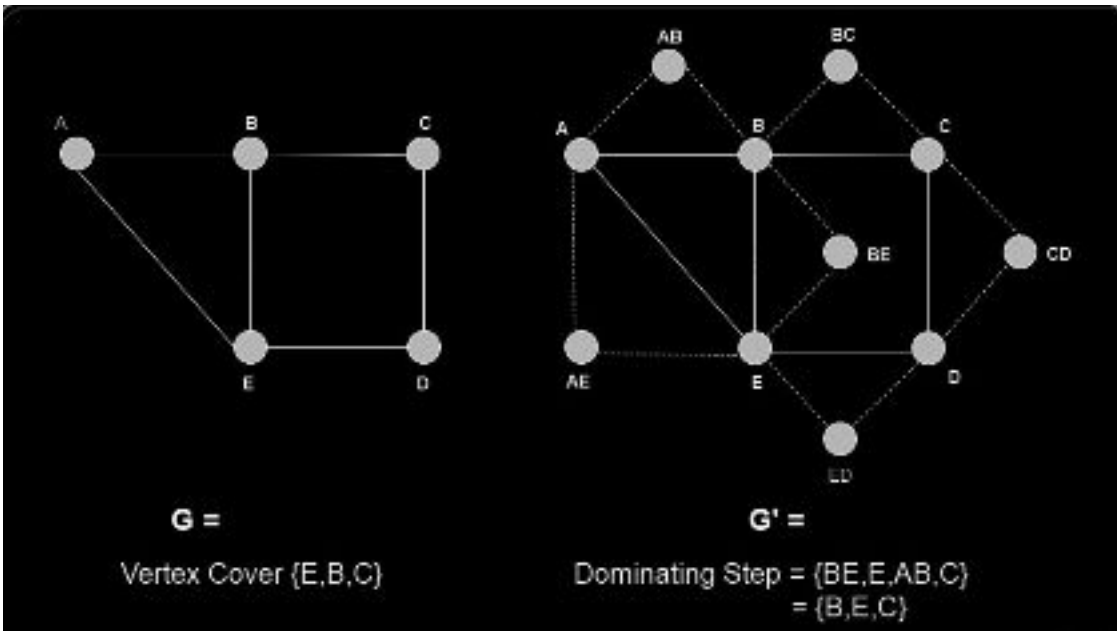
[2] https://www.geeksforgeeks.org/proof-that-dominant-set-of-a-graph-is-np-complete/

The Minimum Dominating Set problem is NP-Hard. To show that MDSP is NP-Hard, at least one of the known NP-Complete problems must be reduced to our problem. For this purpose, we can reduce the Vertex Cover problem to the Minimum Dominating Set Problem. Main issue in the Vertex Cover is "every edge in the original graph is incident to a vertex in the vertex cover set", whereas the main issue in the minimum dominating set is "every vertex in original graph is either member of dominating set or is adjacent to a vertex in dominating set". Hence, we need to show a transformation such that an incident edge in the vertex cover set has a correspondence to an adjacent vertex in the minimum dominating set. In order to do that, we can apply the following idea. Let us assume that we add a vertex into the middle of each edge in the graph. This can be done in polynomial time, namely O(V+E).

For each new added vertex *w* in between vertices *u* and *v* in G', at least one of its adjacent vertices must be in Vertex Cover of G because we need to cover the edge between *u* and *v* in the original graph in order to obtain VC. If we do not include either *u* or *v*, we cannot cover the edge between them. Therefore, for each vertex *x* in the new graph G' is either a member of the VC of the G or adjacent to a member of the VC of G.

After finding the dominating set of G', it is equal to the VC of the G with discarding isolated vertices because isolated vertices do not have any edges to be covered, thus they cannot be in VC. The dominating set of G' may include both vertices from G or newly added vertices. In the second case, among two adjacent vertices of this newly added one, we need to choose the vertex that will create the VC of G.

In the figure below, we have a graph G that we search for the Vertex Cover. By naming the edges of this graph we obtain the G'. Firstly we found The Dominating Set of the G' which is



= *{BE, E, AB, C}* after that we conserve the existing nodes which are *{E, C}* and among the others we choose the nodes that minimize the size such as *{B}* for *{BE, AB}* as a result that gives as the Vertex Cover of G.[3]

2. Algorithm Description

The minimum dominating set problem is proven to be NP-Complete and this implies that no algorithm can solve the problem in polynomial time. However, we can apply a heuristic - greedy approach in order to find the Dominating Set of a graph with no guarantee of finding the minimum cardinality. For this purpose, we make use of a heuristics that applies the greedy algorithm approach. In this method, the algorithm chooses a vertex with the maximum number of adjacent vertices among uncovered vertices in each iteration. Therefore, it is a greedy approach. If there are multiple vertices that satisfy this condition, then one of them is chosen randomly. Therefore, it is guaranteed after these steps that the subset will dominate the maximum number of nodes possible.

[3] https://www.geeksforgeeks.org/proof-that-dominant-set-of-a-graph-is-np-complete/

Steps to be followed:

1. Initially set *D* to be empty

2. For all vertices, assign the *weights* value as the degree of the vertex + 1 (for itself) and declare their *covered* value as false

3. Select a vertex with the maximum *weight* value. If there are more than one, choose it randomly and call it V_i

4. If the maximum value is 0, go step 8**;** else, continue to step 5

5. Add the selected vertex to the set *D*

6. For each vertex V_j that is adjacent to the selected vertex, decrement its *weight* by 1 because one of its adjacent vertices (the chosen one) is covered

i. If the vertex is not covered before, then assign its *covered* value to true and decrement its *weight* by 1 (for itself V_j)

ii. For each vertex V_k that is adjacent to V_j, decrement its *weight* by 1 if its *weight* greater than 0, since V_j is covered

7. Go to step 3

8. The result set *D* is the Minimum Dominating Set

Below is shown more formal description of the heuristic algorithm[4]:

ChooseVertex(Vector ***weight***)

1. $M = max_{1 \le i \le n} weight_i$
2. if $M = 0$ return -1
3. else
4. $\quad S = \{v_i | weight_i = M\}$
5. $\quad$ Return element of S chosen uniformly at random.

Adjust weights after adding v_i to D

AdjustWeights(Graph G, Vector ***weight***, Vector ***covered***, Vertex v_i)

1. $weight_i = 0$
2. for each neighbor v_j of v_i such that $weight_j > 0$
3. $\quad$ if not $covered_i$ decrement $weight_j$
4. $\quad$ if not $covered_j$
5. $\quad\quad$ $covered_j$ = true
6. $\quad\quad$ decrement $weight_j$
7. $\quad\quad$ for each neighbor v_k of v_j
8. $\quad\quad\quad$ if $weight_k > 0$, decrement $weight_k$
9. $covered_i$ = true

Greedy(Graph G)

1. $D = \emptyset$
2. For all vertices v_i
3. $\quad$ $weight_i = 1 + degree(v_i)$
4. $\quad$ $covered_i$ = false
5. Do
6. $\quad$ v = **ChooseVertex**(***weight***)
7. $\quad$ if $v \neq -1$
8. $\quad\quad$ add v to D
9. $\quad\quad$ **AdjustWeights**(G, ***weight***, ***covered***, v)
10. Until $v = -1$
11. Return D

[4] Sanchis, L. A. (2002). Experimental Analysis of Heuristic Algorithms for the Dominating Set Problem. *Algorithmica*, *33*(1), 3–18. https://doi.org/10.1007/s00453-001-0101-z

Ratio Bound: The worst case of the algorithm is better than choosing random vertices. Here is the ratio bound.[5]

$$\frac{|D|}{|D^*|} \leq ln\Delta + 2$$

Δ : is the maximal degree of G

The algorithm always finds a dominating set which does not necessarily have minimum cardinality.

D: Heuristic Solution

D*: Optimum Solution

[5] For further information, please visit (Theorem 7.2):
http://ac.informatik.uni-freiburg.de/teaching/ss_12/netalg/lectures/chapter7.pdf

3. Algorithm Analysis

a) Theoretical Time complexity analysis[6]:

ChooseVertex(Vector ***weight***)
1. $M = max_{1 \le i \le n} weight_i$
2. if $M = 0$ return -1
3. else
4. $\quad S = \{v_i | weight_i = M\}$
5. $\quad$ Return element of S chosen uniformly at random.

O(n)

Adjust weights after adding v_i to D

AdjustWeights(Graph G, Vector ***weight***, Vector ***covered***, Vertex v_i)
1. $weight_i = 0$
2. for each neighbor v_j of v_i such that $weight_j > 0$
3. $\quad$ if not $covered_i$ decrement $weight_j$
4. $\quad$ if not $covered_j$
5. $\quad\quad$ $covered_j$ = true
6. $\quad\quad$ decrement $weight_j$
7. $\quad\quad$ for each neighbor v_k of v_j
8. $\quad\quad\quad$ if $weight_k > 0$, decrement $weight_k$
9. $covered_i$ = true

O(m) at total

Greedy(Graph G)
1. $D = \emptyset$
2. For all vertices v_i
3. $\quad$ $weight_i = 1 + degree(v_i)$
4. $\quad$ $covered_i$ = false
5. Do
6. $\quad$ v = **ChooseVertex**(***weight***)
7. $\quad$ if $v \neq -1$
8. $\quad\quad$ add v to D
9. $\quad\quad$ **AdjustWeights**(G, ***weight***, ***covered***, v)
10. Until $v = -1$
11. Return D

d times

n: the number of vertices in the graph, and that the vertices are numbered $v_0, v_1, \ldots, v_{n-1}$

m: the number of edges in the graph

d: the cardinality of the dominating set

[6] Sanchis, L. A. (2002). Experimental Analysis of Heuristic Algorithms for the Dominating Set Problem. *Algorithmica*, *33*(1), 3–18. https://doi.org/10.1007/s00453-001-0101-z

Using an adjacency list implementation for the graph, it can be checked that each call to ChooseVertex takes O(n) time, and that the total time spent in AdjustWeights throughout execution of the algorithm is O(m). If d is the size of the dominating set found by the algorithm, the total complexity of the algorithm is $\Theta(nd + m) \leq O(n^2)$.

b) Correctness

The Greedy Algorithm above works correctly. Firstly, weight and covered values are assigned for each vertex as initialization. Choose Vertex selects a vertex with a maximum weight if there is any vertex with weight greater than 0. If there is an uncovered vertex, its weight has to be at least one (greater than 0) because of itself, causing the algorithm to run until all vertices are covered. It also makes sure that if all of the vertices have weight equal to 0, then no vertex can be chosen further. AdjustWeights guarantees that the chosen vertex will have a weight=0 and this also prevents ChooseVertex from selecting the chosen vertex which is already added to the dominating set again. Additionally, for each vertex in the dominating set, the neighbours are also assigned as covered. The result in the dominating set is guaranteed to be a dominating set, even though it may not be a minimum. Hence, the algorithm works correctly.

4. Experimental Analysis of The Performance

a) Running Time Experimental Analysis

Beside the theoretical complexity analysis of the algorithm, it is needed to analyze experimental running time of the algorithm. In order to analyze experimentally , we utilized the functions below to calculate standard deviation and confidence intervals.

1) **Standard Deviation:** Mean of the sample input running times (millisecond) are used to calculate standard deviation.

$$\mathrm{SD} = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

```
double calculateSD(vector<double> & data, double & avg) {
    double std = 0.0;
    for (int i=0; i<data.size(); i++) {
        std += pow(data[i]-avg, 2);
    }
    std = sqrt(std/data.size());
    return std;
}
```

2) **Standard Error:** Using standard deviation, standard error is calculated as it is shown below.

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

```
double stderr = stdv/sqrt(times.size());
```

3) **Confidence Interval:** Lastly, confidence intervals are calculated providing 90% and 95% confidence level. The t-values of the levels are 1.645 and 1.96 respectively.

```
void confidenceLevels(vector<double> & data, double & avg, double & stderr) {
    const double val90 = 1.645;
    const double val95 = 1.96;
    double upperMean90 = avg+val90*stderr;
    double lowerMean90 = avg-val90*stderr;
    double upperMean95 = avg+val95*stderr;
    double lowerMean95 = avg-val95*stderr;
    cout << "90% Confidence Interval : " << upperMean90 << " - " << lowerMean90 << endl;
    cout << "95% Confidence Interval : " << upperMean95 << " - " << lowerMean95 << endl;
}
```

Outputs of the above code give the intervals of the running time in terms of milliseconds both at 90% and 95% confidence level.

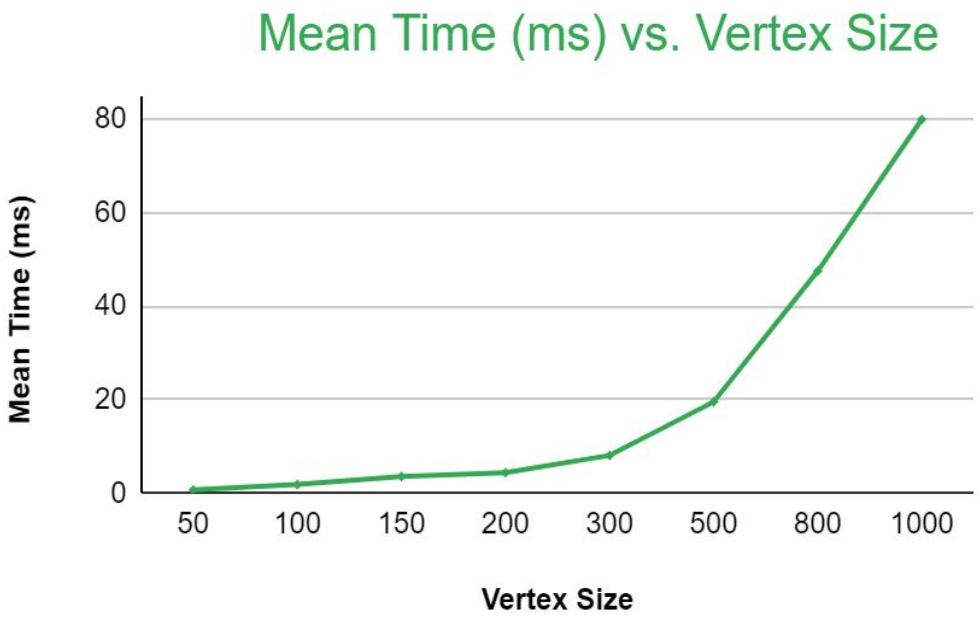
b) Performance Testing

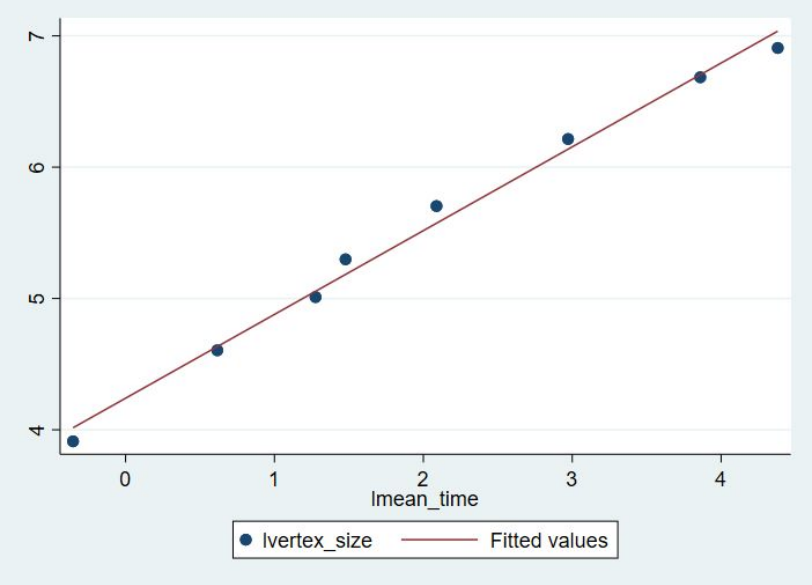
We have graphs G(V, E) with V number of vertices and E number of edges. We have executed the codes on a device that has an Intel i7-7500 64-Bits CPU, 8 GB RAM and Windows 10 operating system. Since the time complexity depends on the vertex size, minimum dominating set cardinality and edge size, we experimented by keeping edge size constant and vertex size constant separately. Running times statistics are experimented and the collected data demonstrated below. In most of the cases, the running time is not linear. The measured running times of the greedy heuristic algorithm are given approximately polynomial time complexity. Therefore, we have converted the results into a **log-log** plot in order to fit a linear line to the graphs and capture more accurate estimations.

A) Keeping Edge Size constant

1) | E | = 1000 : Code is run **100** times per each vertex size

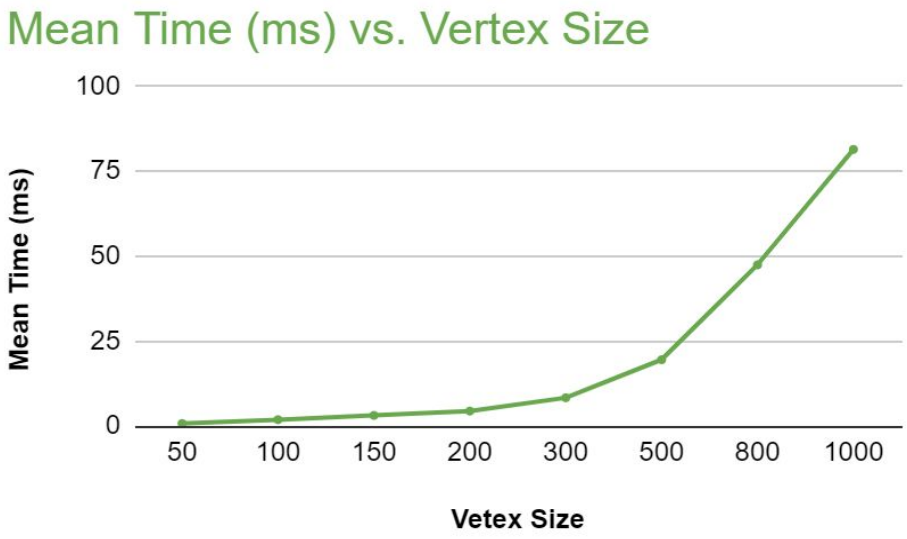
Vertex Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
50	0.702859	0.146898	0.0146898	0.678694 - 0.727024	0.674067 - 0.731651
100	1.85197	0.226659	0.0226659	1.7963 - 1.87087	1.78916 - 1.87801
150	3.58277	0.898339	0.0898339	3.43499 - 3.73054	3.40669 - 3.75884
200	4.38184	0.534861	0.0534861	4.29386 - 4.46983	4.27701 - 4.48668
300	8.07283	0.974058	0.0974058	7.91259 - 8.23306	7.88191 - 8.26374
500	19.5339	1.35137	0.135137	19.3116 - 19.7562	19.2691 - 19.7988
800	47.4692	1.62798	0.162798	47.2014 - 47.737	47.1501 - 47.7883
1000	79.8768	5.87595	0.587595	78.9102 - 80.8434	78.7251 - 81.0285

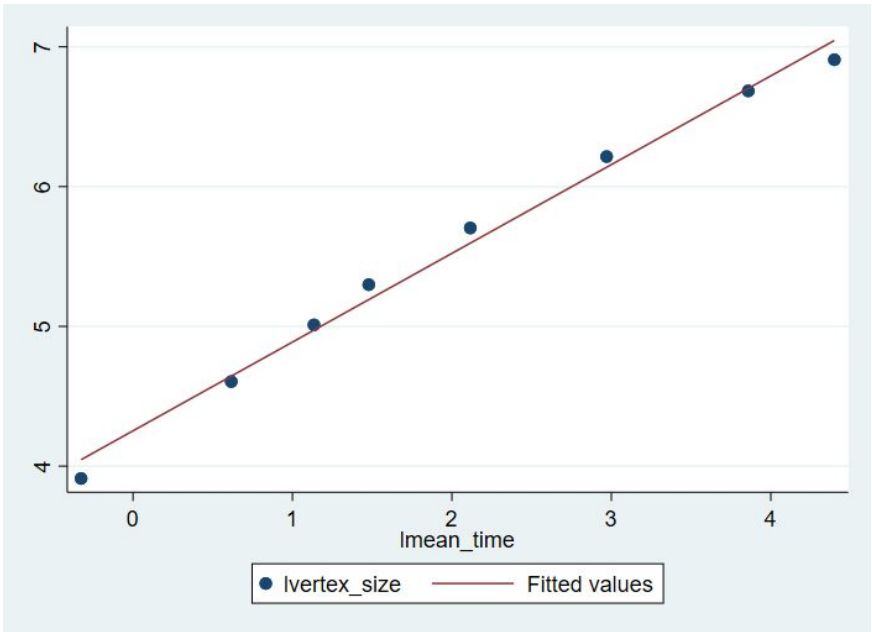




2) | E | = 1000 : Code is run **1000** times per each vertex size

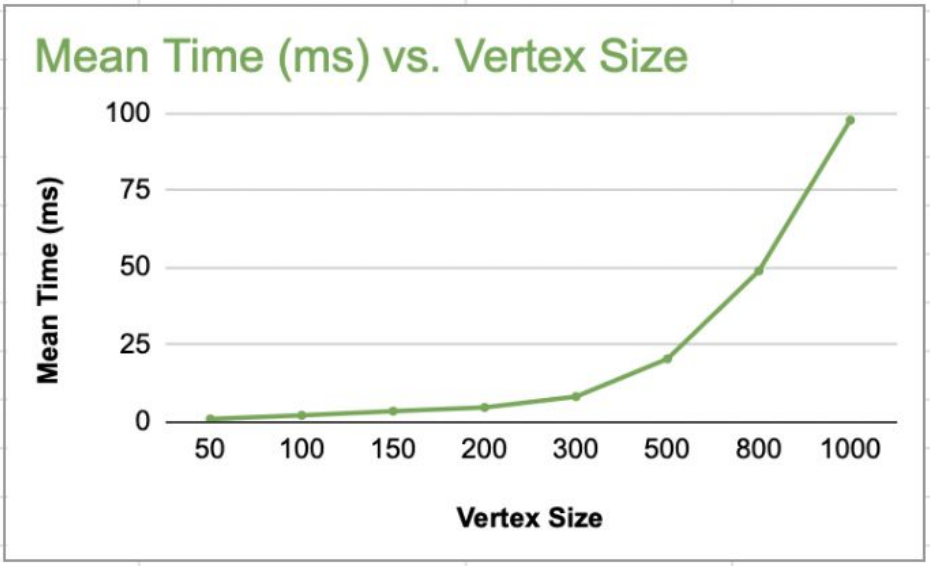
Vertex Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
50	0.722328	0.18797	0.00594415	0.71255 - 0.732106	0.710677 - 0.733978
100	1.85197	0.26733	0.00845372	1.83807 - 1.86588	1.8354 - 1.86854
150	3.11094	0.390069	0.0123351	3.09065 - 3.13123	3.08677 - 3.13512
200	4.38751	0.734024	0.0232119	4.34933 - 4.4257	4.34202 - 4.43301
300	8.2936	3.38057	0.106903	8.11775 - 8.46946	8.08407 - 8.50313
500	19.5003	1.58212	0.050031	19.418 - 19.5826	19.4022 - 19.5983
800	47.4077	2.05067	0.064848	47.301 - 47.5144	47.2806 - 47.5348
1000	81.3638	10.7591	0.340231	80.8041 - 81.9235	80.697 - 82.0307

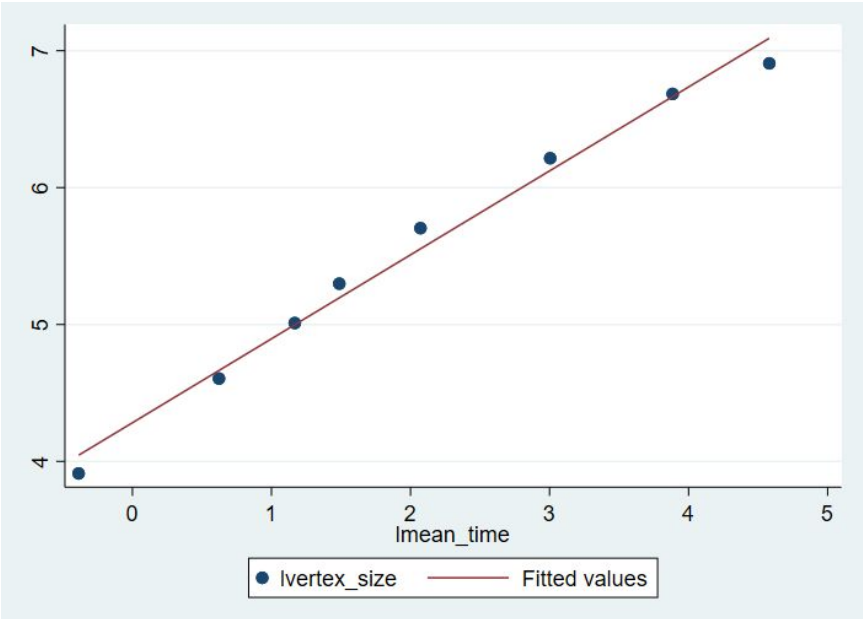




3) E | = 1000 : Code is run **5000** times per each vertex size

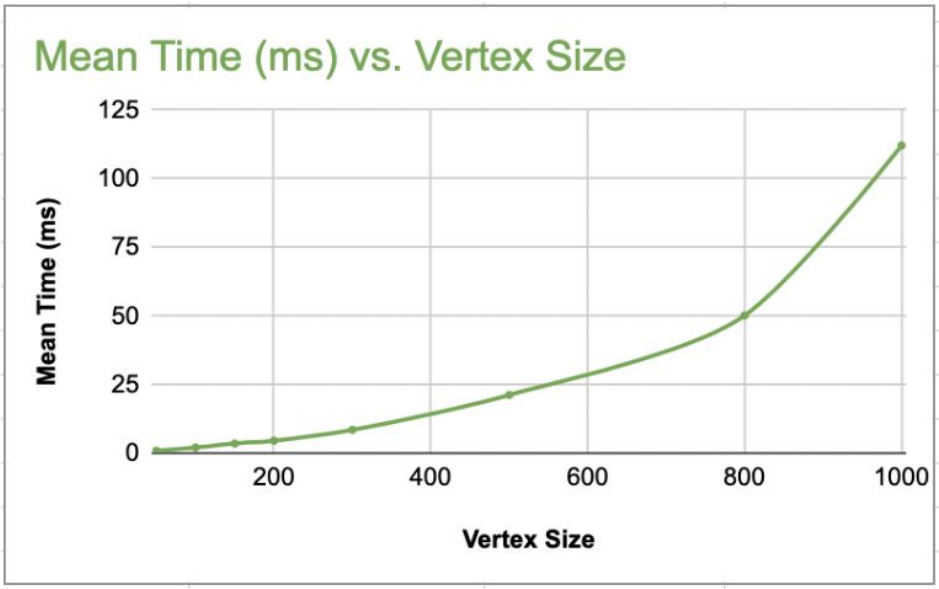
Vertex Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
50	0.679278	0.109236	0.00154483	0.676737 - 0.681819	0.67625 - 0.682306
100	1.86377	0.304011	0.00429937	1.8567 - 1.87085	1.85535 - 1.8722
150	3.21508	0.896496	0.0126784	3.23593 - 3.19422	3.19023 - 3.23993
200	4.42726	0.904919	0.0127975	4.40621 - 4.44831	4.40217 - 4.45234
300	7.93891	0.759393	0.0107394	7.92125 - 7.95658	7.91786 - 7.95996
500	20.1795	1.94841	0.0275547	20.1342 - 20.2248	20.1255 - 20.2335
800	48.6931	6.95672	0.0983829	48.5313 - 48.8549	48.5003 - 48.8859
1000	97.5722	19.7632	0.279493	97.1124 - 98.0319	97.0244 - 98.12

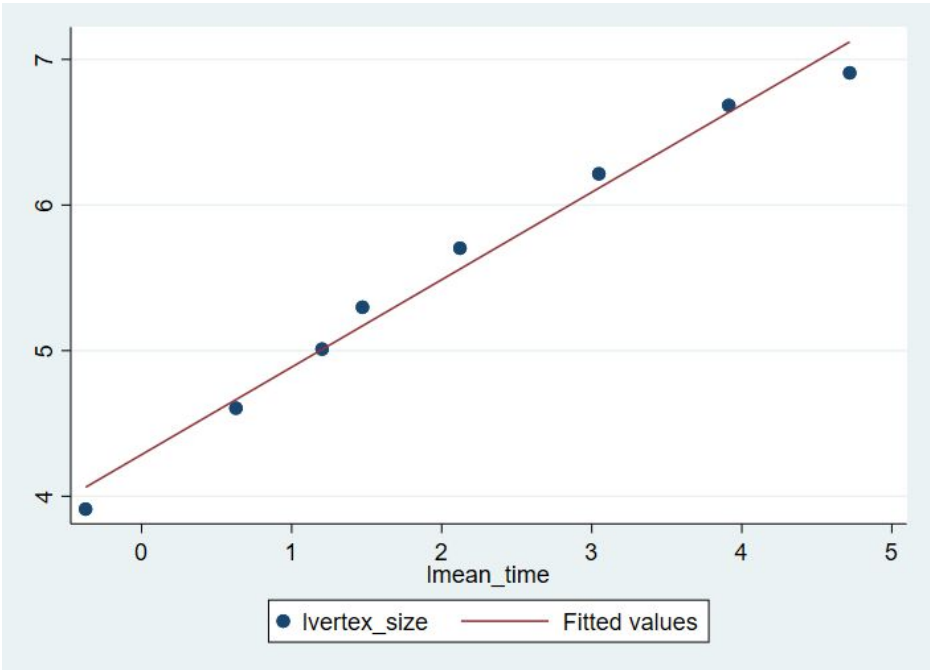




4) E | = 1000 : Code is run **10000** times per each vertex size

Vertex Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
50	0.68854	0.140201	0.00140201	0.686234 - 0.690846	0.685792 - 0.691288
100	1.87523	0.357258	0.00357258	1.86935 - 1.8811	1.86823 - 1.88223
150	3.33006	1.4987	0.014987	3.30541 - 3.35471	3.30069 - 3.35943
200	4.3588	0.540836	0.00540836	4.34991 - 4.3677	4.3482 - 4.3694
300	8.34611	1.13832	0.0113832	8.32739 - 8.36484	8.3238 - 8.36842
500	21.0618	4.34795	0.0434795	20.9903 - 21.1334	20.9766 - 21.1471
800	50.0688	6.84879	0.0684879	49.9561 - 50.1815	49.9346 - 50.203
1000	112.095	33.7603	0.337603	111.54 - 112.65	111.433 - 112.757

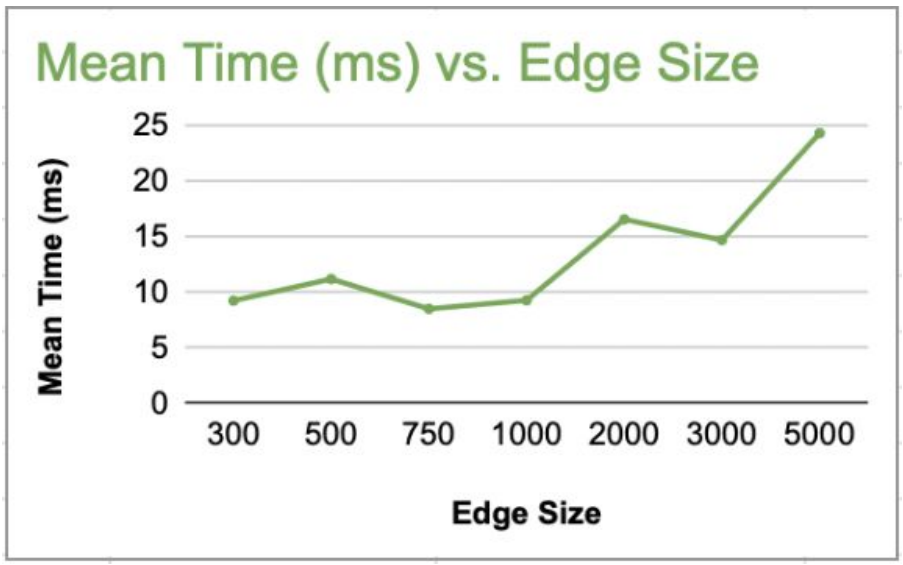


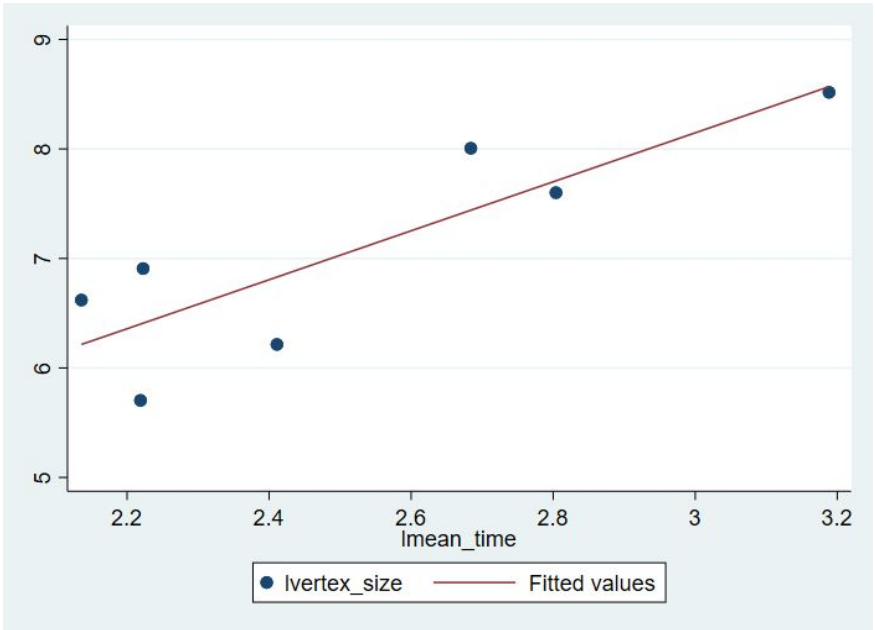


B) Keeping Vertex Size constant

1) | V | = 300 : Code is run **100** times per each edge size

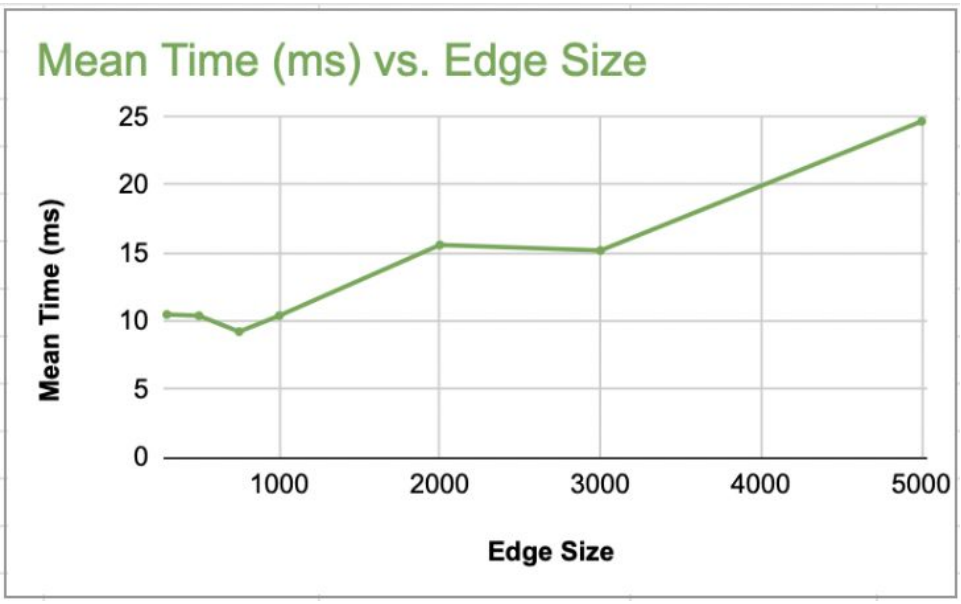
Edge Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
300	9.1975	1.95563	0.195563	8.87579 - 9.5192	8.81419 - 9.5808
500	11.1459	2.3208	0.23208	10.7642 - 11.5277	10.6911 - 11.6008
750	8.46312	0.879953	0.0879953	8.31836 - 8.60787	8.29064 - 8.63559
1000	9.23127	1.57491	0.157491	8.9722 - 9.49034	8.92259 - 9.53995
2000	16.5092	2.66614	0.266614	16.0706 - 16.9478	15.9867 - 17.0318
3000	14.6437	0.660225	0.0660225	14.5351 - 14.7523	14.5143 - 14.7731
5000	24.2497	5.54378	0.554378	23.3377 - 25.1617	23.1631 - 25.3363

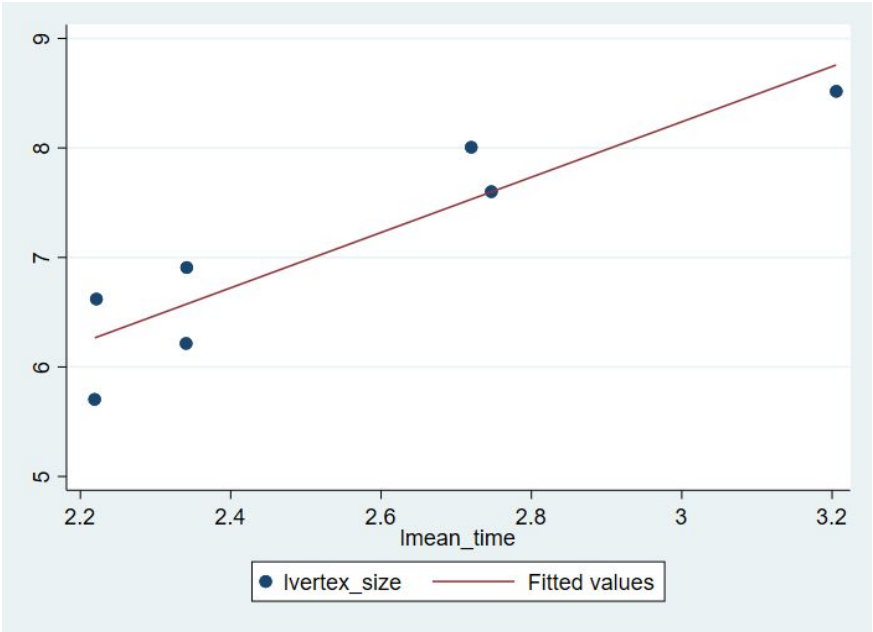




2) | V | = 300 : Code is run **1000** times per each edge size

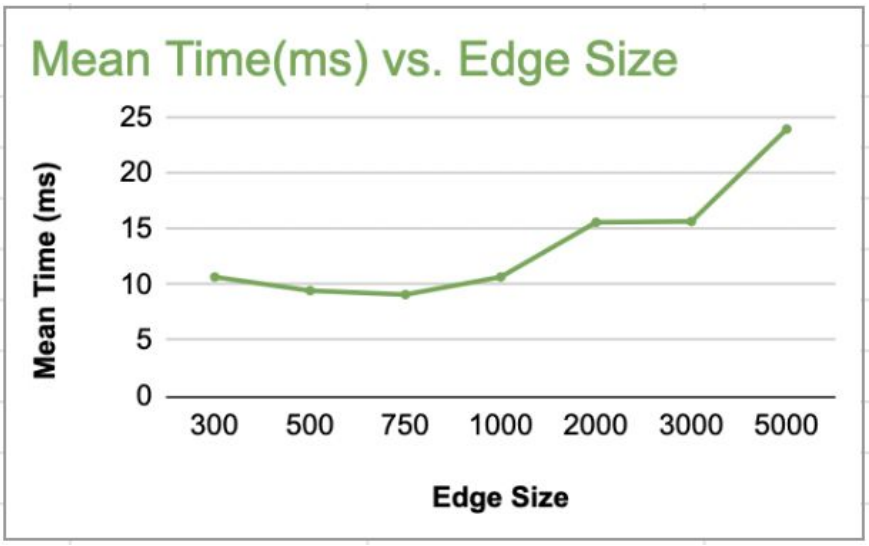
Edge Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
300	10.4815	2.62994	0.083166	10.3447 - 10.6183	10.3185 - 10.6445
500	10.3879	2.5788	0.0815487	10.2537 - 10.522	10.228 - 10.5477 -
750	9.21946	2.91568	0.092202	9.06779 - 9.37113	9.03875 - 9.40018
1000	10.3964	2.05545	0.064999	10.2895 - 10.5034	10.269 - 10.5238
2000	15.591	3.31772	0.104915	15.4184 - 15.7636	15.3854 - 15.7966
3000	15.1808	2.41612	0.0764046	15.0551 - 15.3065	15.0311 - 15.3306
5000	24.6701	5.42139	0.17144	24.3881 - 24.9521	24.3341 - 25.0061

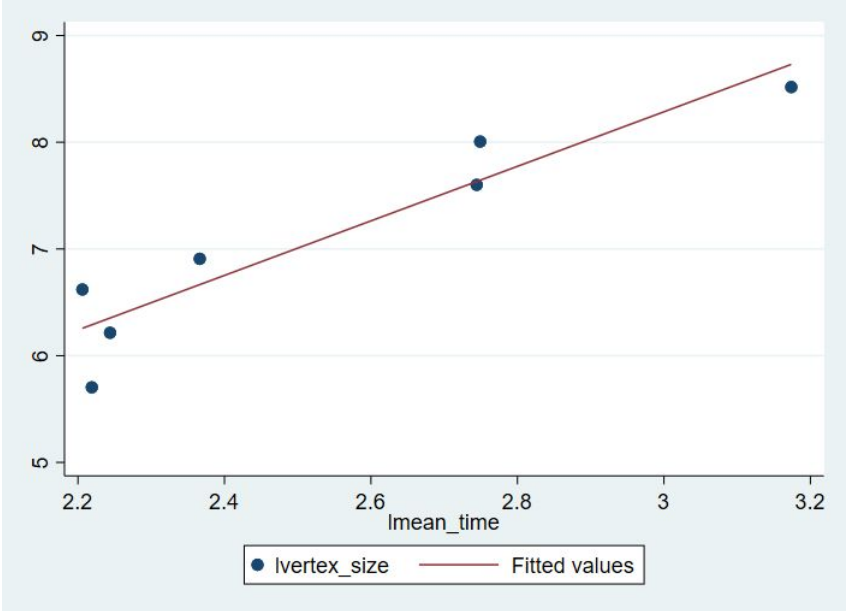




3) | V | = 300 : Code is run **5000** times per each edge size

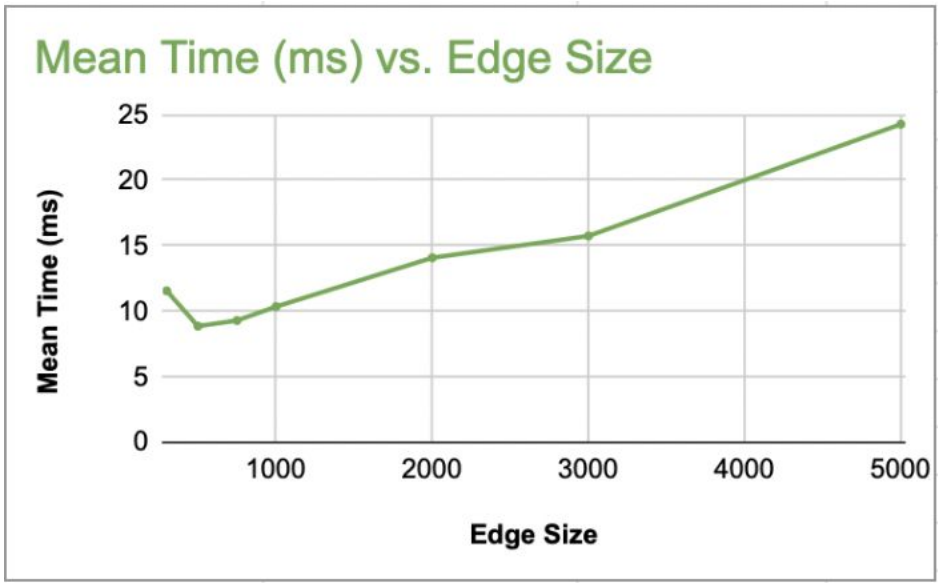
Edge Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
300	10.6544	4.39427	0.0621444	10.5522 - 10.7567	10.5326 - 10.7762
500	9.42971	3.71494	0.0525372	9.34329 - 9.51614	9.32674 - 9.53269
750	9.07973	2.23458	0.0316018	9.02775 - 9.13172	9.01779 - 9.14167
1000	10.6578	2.78504	0.0393864	10.593 - 10.7226	10.5806 - 10.735
2000	15.5558	3.22154	0.0455594	15.4809 - 15.6308	15.4665 - 15.6451
3000	15.6293	3.05135	0.0431527	15.5584 - 15.7003	15.5448 - 15.7139
5000	23.9002	4.09001	0.0578415	23.8051 - 23.9954	23.7869 - 24.0136

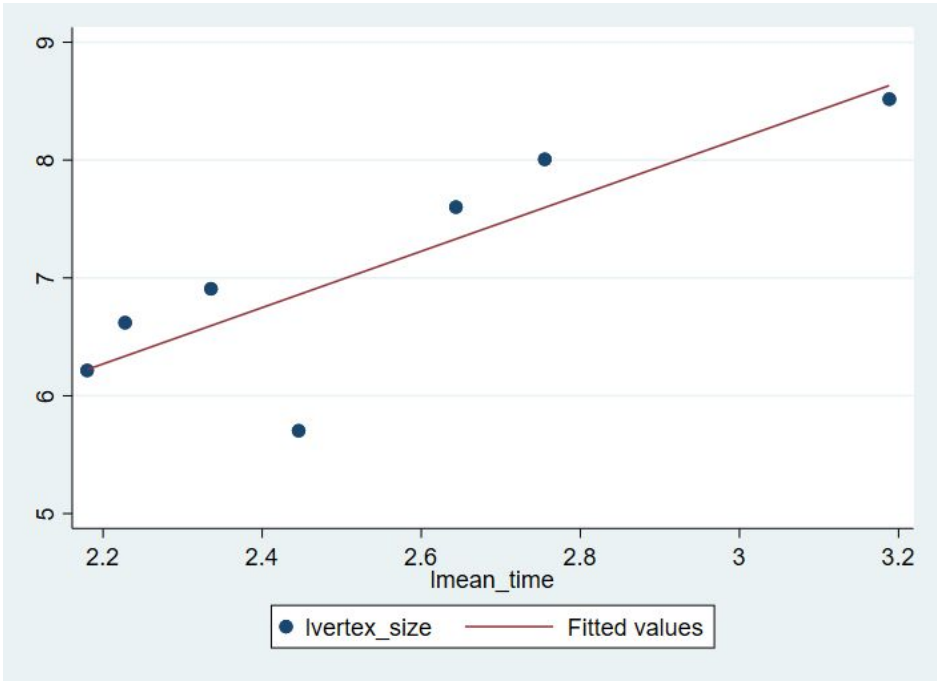




4) | V | = 300 : Code is run **10000** times per each edge size

Edge Size	Mean Time (ms)	Standard Deviation	Standard Error	90% CL	95% CL
300	11.541	5.27388	0.0621444	10.5522 - 10.7567	10.5326 - 10.7762
500	8.84645	3.45187	0.0345187	8.78967 - 8.90324	8.7788 - 8.91411
750	9.27855	2.87386	0.0287386	9.23128 - 9.32583	9.22223 - 9.33488
1000	10.3371	2.35892	0.0235892	10.2983 - 10.3759	10.2908 - 10.3833
2000	14.0634	4.10628	0.0410628	13.9959 - 14.131	13.983 - 14.1439
3000	15.7253	2.38512	0.0238512	15.6861 - 15.7645	15.6786 - 15.7721
5000	24.2502	5.4491	0.054491	24.1606 - 24.3398	24.1434 - 24.357





5. Experimental Analysis of the Correctness

a) Correctness of the Heuristic Algorithm

Heuristic approach may not always give the minimum dominating set. However, the result should be a dominating set.

The function below checks whether the dominating set found by a heuristic greedy algorithm is a dominating set or not.

```
bool VertexSet::isDominatingSet() {
    for (int i = 0; i < graph->get_num_v(); i++) {
        if (set[i] == 0) {
            bool covered = false;
            for (auto it = graph->begin(i); it != graph->end(i); it++) {
                if (set[(*it).get_dest()] != 0) {
                    covered = true;
                }
            }
            if (!covered) {
                return false;
            }
        }
    }
    return true;
}
```

In all of the cases, our algorithm has succeeded to find a dominating set for given graphs which is tested with the code above.

b) Comparison of Exact Algorithm and Heuristics Algorithm

As a brute force approach, we can try all possible combinations of all vertices. However, we have to try $\mathbf{2^v - 1}$ different possible combinations and it takes a really great amount of time and grows exponentially.

Since our algorithm is heuristic, it does not guarantee an optimal solution. On the other hand, it guarantees dominating sets.

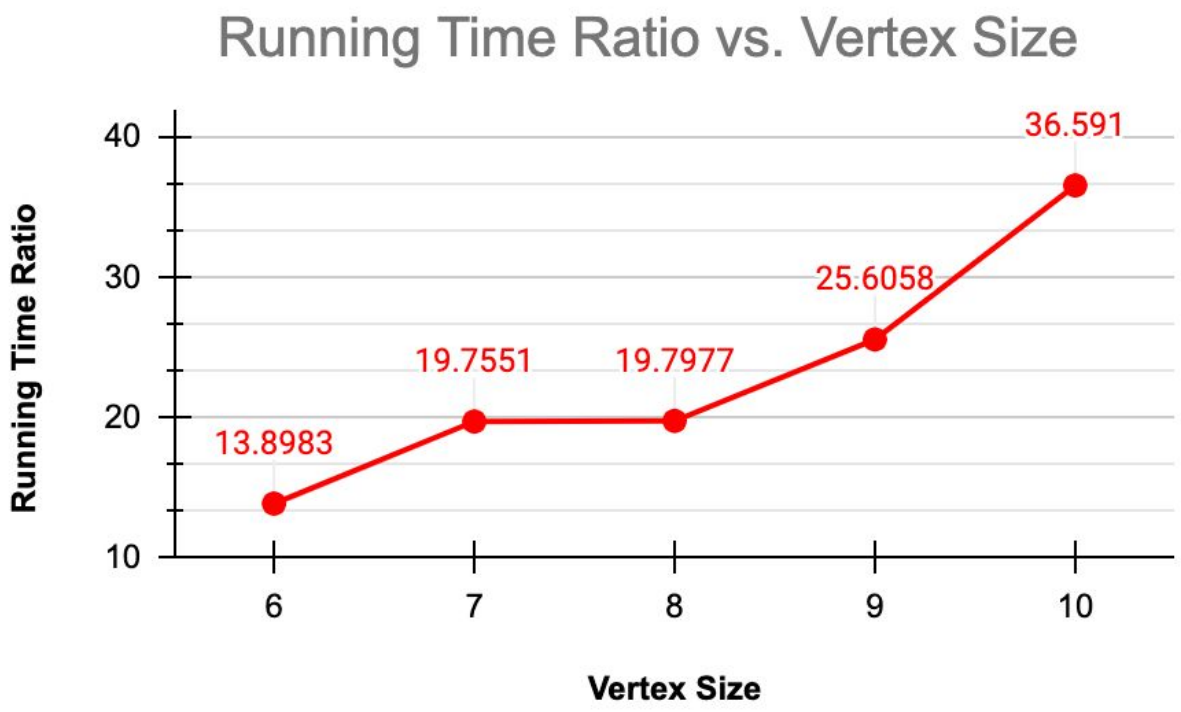
Since the brute-force exact algorithm runs in an exponential time, small inputs are tested both in exact algorithm and heuristic algorithm. We created **2000 random graphs** per each vertex size and in total we tested 10.000 different inputs. 2 new metrics are 'Running times ratio' and 'correctness ratio' are compared for each vertex size 6,7,8,9 and 10.

Here is the code that we generated the random graphs.

```
void generate_black_box(int vertex, int edge) // number of vertices and edges are given
{
    srand(time(NULL));

    // an adjacency matrix to keep edges
    // rows represent vertices - colums represent edges
    vector<vector<int>> mat(vertex, vector<int> (vertex));

    int edge_counter = 0;

    // until we created desired number of edges
    while (edge != edge_counter)
    {
        // generate 2 vertices
        int random_v1 = rand() % vertex;
        int random_v2 = rand() % vertex;

        // put an edge between them
        if(mat[random_v1][random_v2] == 0 && random_v1 != random_v2)
        {
            // undirected graph
            mat[random_v1][random_v2] = 1;
            mat[random_v2][random_v1] = 1;

            edge_counter++;
        }
    }
}
```

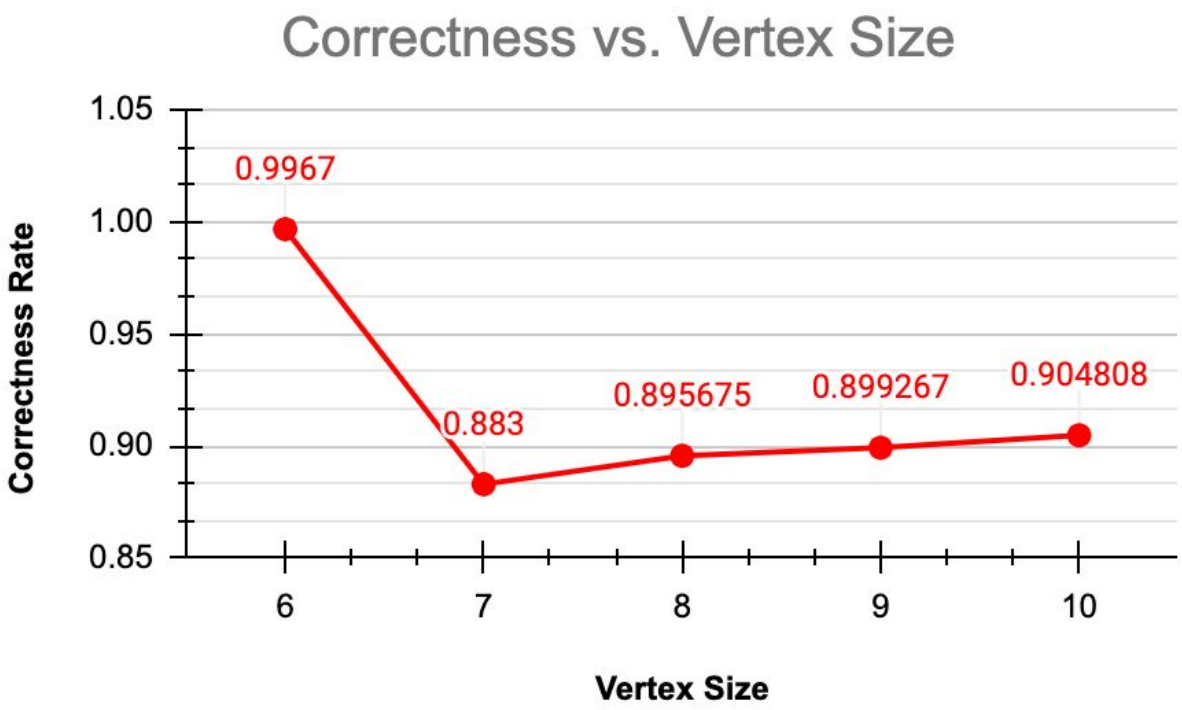
Below the comparison results are shown. Note that it is black box testing and it covers all edge cases such as isolated vertex.

Graph 1: Running time ratio = **time** for exact algorithm / **time** for the heuristic algorithm



It shows that running time of the exact algorithm is at least 13 times larger than the running time of the heuristic algorithm. The graph indicates that the rate is growing as the number of vertices increases.

Graph 2: Correctness = **cardinality** of exact algorithm / **cardinality** of heuristic algorithm

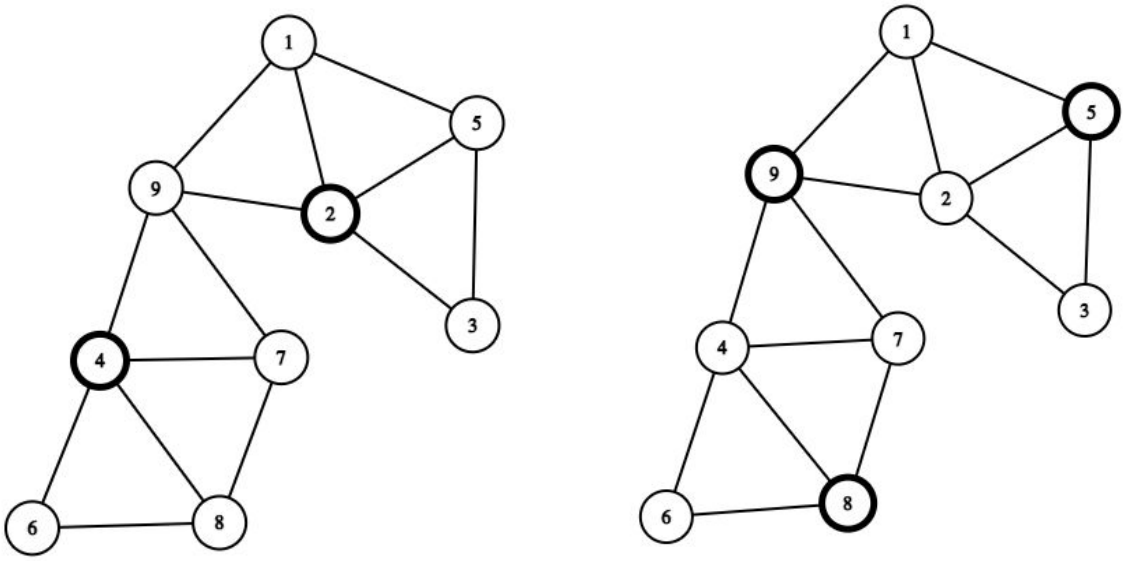


The graph is computed as (cardinality of exact algorithm / cardinality of heuristic algorithm). It shows the quality of the heuristic algorithm in terms of finding a minimum dominating set. Quality = (Correctness Rate for all vertex size / 5) = 0.915. The graph indicates that using a heuristic algorithm is a reasonable decision while we are solving this problem because it finds a dominating set with approximately 0.085 error rate.

Functional Testing

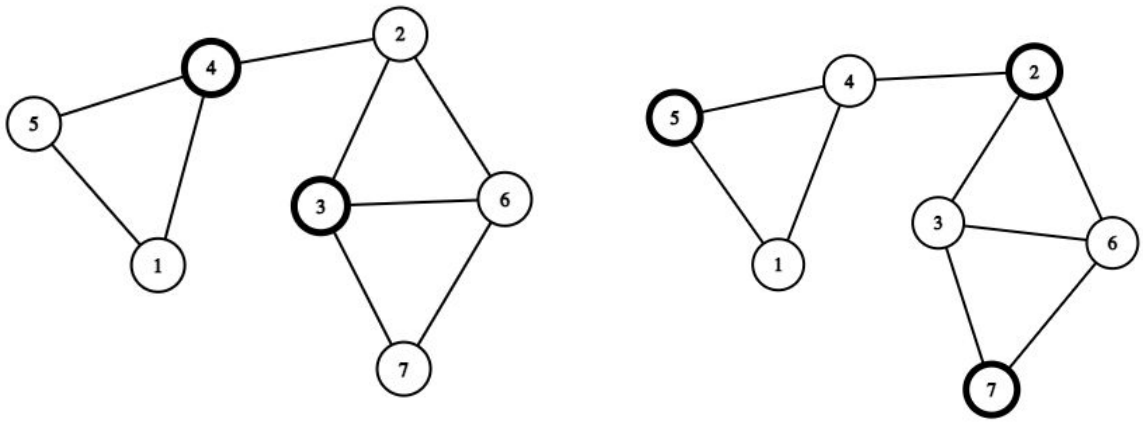
Some of the failed and succeeded cases are visualized. Bold ones represent the vertices in the minimum dominating set.

FAILED



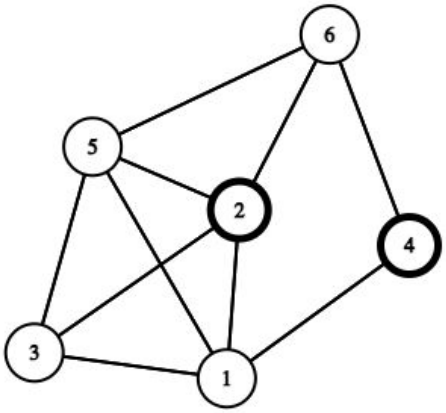
Left one is an exact-optimal solution, the cardinality is 2, while the right one is the result of greedy-heuristic algorithm, the cardinality is 3.

FAILED



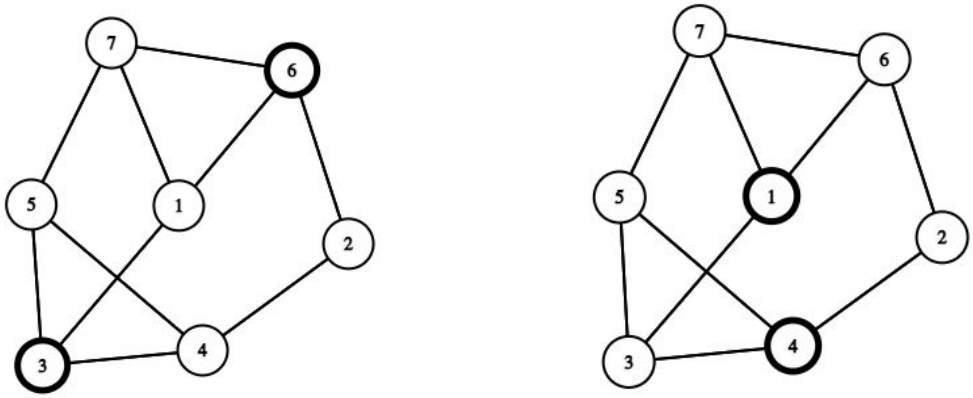
The left one is an exact-optimal solution with the cardinality 2. The right one is the result of a heuristic-greedy algorithm with cardinality 3.

SUCCEEDED



This is an optimal solution that the heuristic algorithm and the exact algorithm finds correctly.

SUCCEEDED



The left one is an optimal solution with the cardinality 2. The right one is the result of our algorithm with cardinality 2 and it is also an optimal solution.

6. Discussion

The subject of this project was the Minimum Dominating Set problem and this problem has been proven to be NP-complete. It is shown that MDSP could not be solved in polynomial time and could be reduced into Vertex Cover problem which is also an NP-Complete problem. There are many heuristic algorithms that solve this problem in polynomial time. However, these approaches do not guarantee to find optimal solutions. The greedy heuristic approach has been used in this project and this approach does not guarantee to find a minimum dominating set, whereas it guarantees to find a dominating set. The heuristic method is picking a vertex which has the highest degree among uncovered vertices in each iteration, until all vertices are dominated. The time complexity of the algorithm is polynomial which is better than exponential. Furthermore, the theoretical ratio bound is found ln(maximal degree of G) + 2. Experimental running time analysis of heuristic (O(nd+m) theoretically) is calculated with statistics of the running time including mean, standard deviation, standard error, 90% and 95% confidence intervals. The results are demonstrated in charts. Additionally, experimental analysis of correctness is done by starting with assessing the correctness of the heuristic approach that the result found whether is a dominating set or not. Secondly, the results of the both heuristic and exact approach is compared with using charts. Lasty, we have done functional testing on our heuristic algorithm by using Black Box Testing method and generating random graphs. We have provided some of the failed and the successful cases with explanations. It is shown that the quality of the heuristic algorithm is 0.915 and it seems reasonable. Since there is a trade-off between time and the correctness, the heuristic algorithm we experimented on is faster and can be accepted as a result most of the time with a quality of 91.5% correctness.

References

1ndrew100, “graph_theory_smallest_dominating_set”, *Retrieved January 26, 2021 from* https://github.com/1ndrew100/graph_theory_smallest_dominating_set

GeeksForGeeks.com, “Proof that Dominant Set of a Graph is NP-Complete”, *Retrieved December 8, 2020 from* https://www.geeksforgeeks.org/proof-that-dominant-set-of-a-graph-is-np-complete/

Mount, “NP-Completeness: Clique, Vertex Cover, and Dominating Set”, *Retrieved December 1, 2020 from* http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect21-np-clique-vc-ds.pdf

Sanchis, L. A. (2002). “Experimental Analysis of Heuristic Algorithms for the Dominating Set Problem”. *Algorithmica*, *33*(1), 3–18. https://doi.org/10.1007/s00453-001-0101-z

Otávio Augusto, “DominatingSetHeuristics”, *Retrieved January 2, 2021 from* https://github.com/oaugusto/DominatingSetHeuristics

Zhang et. al., “Proof of NP-completeness of Dominating Set problem”*, Retrieved December 1, 2020 from* https://cs.ubishops.ca/home/cs467/more-np-complete/zhang-dominating-set.pdf