CS301

2022-2023 Spring

Project Report

Group 061

::Group Members::

Nisa Erdal 28943

Berkay Barış Turan 28132

1. Problem Description

Given an undirected graph G = (V, E), a subset of vertices D ⊆ V is a dominating set if, for every vertex u ∈ V - D, there is a vertex v ∈ D such that (u, v) ∈ E. The domination number of the graph is $\gamma(G) := \min\{|D| : D \text{ is a dominating set of } G\}$. In other words, a dominating set for a graph G is a subset D of its vertices, such that any vertex of G is either in D or has a neighbour in D. The minimum dominating set problem focuses on finding the dominating set with the minimum number of vertices in it. Dominating sets are used in various applications, such as optimizing the placement of sensors or other devices in a network to detect or control events. By identifying the minimum set of sensors that can cover the entire network, a dominating set can help reduce the cost and complexity of network design. However, finding the minimum dominating set for an undirected graph is believed to be computationally intractable in the worst case. The minimum dominating set problem is NP-complete for undirected graphs, meaning that there is no known algorithm that can solve this problem efficiently in polynomial time for all graphs (Garey & Johnson, 1979). The proof of this theorem shows that the problem is reducible to the vertex cover problem and is both NP-hard and in NP, therefore NP-complete (Garey & Johnson, 1979).

2. Algorithm Description

a. Brute Force Algorithm

The minimum dominating set of a graph G = (V, E) can be found by generating all the subsets of G, testing each of them on the dominating set criteria, and selecting the minimum subset of G which fits the criteria. Testing if a given subset is a dominating set or not can be done by checking if all the vertices of the graph outside of the subset have an adjacent vertex in the subset. If this condition holds, the subset is one of the dominating sets of the graph. Each subset of the graph is tested by this condition and the dominating set with minimum size is updated when necessary, then returned. The algorithm takes $O(2^n*k*(n-k))$ times in the worst case, which is an exponential not efficient time. The pseudo-code of the algorithm is:

1. Generate all subsets of G:

```
subsets = []
for k = 0 to n:
        for subset in subsets
                new_subset = subset + [i]
subsets = subsets + new_subset
```

2. For each subset in subsets, check if it is a dominating set:

```
min = INFINITY
for subset in subsets:
        if is_dominating_set(G, subset) and subset size < min:
                min_dom_set = subset
Return min_dom_set
4. Function is_dominating_set(G, subset):
uncovered_nodes = set(range(n)) - set(subset)
for node in uncovered_nodes:
        if not any(G[node][j] == 1 for j in subset):
                return False
return True
```

b. Heuristic Algorithm

To find an approximation algorithm for the smallest dominating set problem, a sequential greedy algorithm is used.

1. S := ∅;
2. while ∃ white nodes do
3. choose v ∈ {x | w(x) = max u∈V {w(u)}};
4. S := S U {v};
5. for vertex in range(num_vertices):
6. if adj_matrix[max_weight_vertex][vertex] == 1:
7. white_vertices.discard(vertex)
8. end while

It starts with an empty set 'S' and iteratively selects a vertex 'v' with the highest weight (where the weight function 'w' assigns weights to each vertex) among the remaining white (not yet selected) vertices. The selected vertex 'v' is added to the set 'S', and the process continues until there are no more white vertices left. White vertices are updated in each step, discarding the neighbours of the selected vertex. The weights are assigned to each vertex as the number of vertices adjacent to the vertex. The algorithm used to compute the weight function w(u) is below:

```
function assign_weights_degree(adj_matrix):
   num_vertices := length(adj_matrix)
   weights := array of size num_vertices
   for vertex in range(num_vertices) do
      degree_sum := 0
      for neighbour in adj_matrix[vertex] do
         degree_sum := degree_sum + neighbour
      end for
      weights[vertex] := degree_sum
   end for
   return weights
end function
```

The Greedy Algorithm provides an approximation of (ln Δ + 2), where Δ represents the maximum degree in the graph (Kuhn, 2013). In other words, if S is the dominating set computed by the algorithm and S∗ is the optimal dominating set, then the size of S divided by the size of S∗ is less than or equal to ln Δ + 2.

To prove this theorem, we analyse the algorithm's steps. Each time a new node is added to the dominating set (greedy step), it incurs a cost of 1. Instead of assigning the entire cost to the added node, we distribute it equally among all the newly covered nodes. Let's assume that the chosen node v, in line 3 of the algorithm, is white and its white neighbours are v1, v2, v3, and v4. In this case, each of the 5 nodes (v, v1, v2, v3, v4) is charged with 1/5 of the cost. If v is chosen as a gray node, only v1, v2, v3, and v4 are charged (each with 1/4).

Now, assuming we have knowledge of an optimal dominating set $S*$. Based on the definition of dominating sets, for each node not in $S*$, we can assign a neighbour from $S*$. By assigning each node to exactly one neighbouring node in $S*$, the graph can be decomposed into stars, where each star has a dominator (a node in $S*$) as its center and non-dominators as leaves. It is clear that the cost of an optimal dominating set is 1 for each such star. In the following explanation, we demonstrate that the amortised cost (distributed costs) of the greedy algorithm is at most $\ln \Delta + 2$ for each star. This demonstration is sufficient to prove the theorem.

Consider a single star with the center $v*$ belonging to $S*$ before choosing a new node u in the greedy algorithm. The number of nodes that become dominated when u is added to the dominating set is denoted as w(u). Thus, if any white node v in the star of $v*$ becomes gray or black, it incurs a charge of 1/w(u). According to the greedy condition, u is a node with the maximum span, and therefore $w(u) \geq w(v*)$. Consequently, v is charged at most $1/w(v*)$. Once a node becomes gray, it is not charged anymore.

Hence, the first node that is covered in the star of $v*$ incurs a charge of at most $1/(d(v*) + 1)$, where $d(v*)$ represents the degree of $v*$. When the second node is covered, it incurs a charge of at most $1/d(v*)$, given that $w(v*) \geq d(v*)$. In general, the i^{th} node covered in the star of $v*$ incurs a charge of at most $1/(d(v*) + i - 2)$ (Kuhn, 2013).

3. Algorithm Analysis

a. Brute Force Algorithm

Theorem: Considering all subsets of a given graph and selecting the minimum-sized subset from all found dominating sets as the minimum dominating set gives the exact solution to find the minimum dominating set for all graphs.

The correctness of the theorem can be shown as follows:

Let G = (V, E) be a graph, and let D be a dominating set of G. By definition of the dominating sets, D must be one of the $2^{|V(G)|}$ subsets of G since D consist of the vertices of G. Considering all subsets of a given graph guarantees to find the minimum dominating set since the sizes of the subsets that pass the dominating set test are compared and the minimum sized dominating set is returned. This theorem also works for the graphs with no edges since in this case, no dominating set would be found until testing the subset of the graph which is all vertices of the graph itself. Therefore, testing all subsets of a given graph until finding the minimum sized dominating set is a valid solution for guaranteed correct result.

The running time of the algorithm can be computed by considering all the steps used in the algorithm. Firstly, finding all the subsets of G requires $\boldsymbol{\Theta}(2^n)$ time since there are 2^n subsets of any n-sized set and each subset can be generated in constant time. For any subset, the condition of being a dominating set can be tested in the $\boldsymbol{\Theta}(k*(n-k))$ time where k is the number of vertices of the graph outside of the subset and algorithm iterates over this k vertices n - k times to see the adjacency. Testing all 2^n subsets on the dominating set condition takes $\boldsymbol{\Theta}(2^n*k*(n-k))$ time. Therefore, the algorithm takes $\boldsymbol{\Theta}(2^n*k*(n-k) + 2^n)$ times in the worst case.

b. Heuristic Algorithm

Theorem: The algorithm using the greedy approach described above correctly computes an approximately optimal sized dominating set for an undirected graph.

Proof: Let G = (V, E) be an undirected graph, where V represents the set of vertices and E represents the set of edges. The algorithm initialises an empty set S. It is assumed that w(u) represents the weight assigned to vertex u, which is computed using the assign_weights_degree function as described in the algorithm. To show that the algorithm correctly computes a dominating set, S, for G; suppose, by contradiction, that S is not a dominating set for G. This implies that there exists at least one vertex, v, in V that is not dominated by any vertex in S. During each iteration of the while loop, the algorithm selects a vertex, v, with the maximum weight w(v) from the set of unprocessed vertices (white nodes). By selecting v and adding it to S, it is guaranteed that v is included in the dominating set, as every vertex adjacent to v becomes dominated. Since S is updated by adding v, v is removed from the set of unprocessed vertices (white nodes). This process continues until all vertices have been processed. By the end of the algorithm, every vertex in G is either in S or adjacent to a vertex in S. Hence, S is a dominating set for G. Moreover, the approximation relation of the S with the optimal dominating set S* is shown in part 2b. Thus, the theorem is proven.

Finding the weights of each vertex takes O(V2) time since the sum of each row is calculated for finding the number of adjacent vertices to each vertex. Finding the dominating set of the graph requires one while loop and two nested for loops inside the while loop. The while loop iterates over the white vertices, the number of white vertices is O(V) in the worst case. Finding the vertex with the maximum weight also takes O(V) times since the algorithm again iterates over the remaining white vertices. Then, the last loop iterates over the neighbours of the selected vertex. The maximum number of the vertex is V-1 therefore this loop also takes O(V) time. Total cost of the function of finding the dominating set becomes O(V*(V+V)). Worst case complexity of the algorithm is $O(V^2)$.

4. Sample Generation (Random Instance Generator)

The matrix representation being utilized in the project is the adjacency matrix which constitutes binary elements where 0's are denoting no established edge between the corresponding nodes whereas 1's denoting an established edge connection. In addition to this, the instances are generated as undirected graphs as suggested for this problem. Step by step explanation of the random adjacency matrix generator algorithm is as follows:

1. Input: n (i.e., number of nodes)
2. Create n x n matrix filled with random 0's and 1's.
3. Set the diagonal elements to 0 to ensure that no vertex is connected to itself.
4. Set the (i, j) and (j, i) elements to be the same value to ensure that the graph is undirected.
5. Return the resulting matrix.

Only parameter required is the number of nodes constituting the graph. Any other property of the graph is not specified such as whether the graph is connected, or it has connected component since we want our algorithm to work on both cases.
The pseudocode of the random instance generator we utilized is provided below.

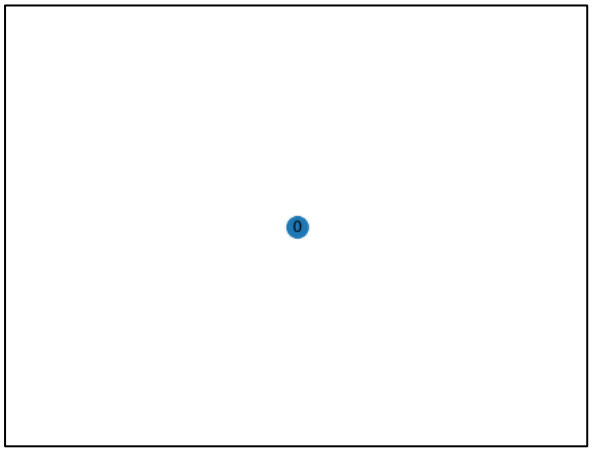
```
function random_adjacency_matrix(n)
    matrix = n x n matrix filled with random 0's and 1's
    for i = 0 to n-1 do
        matrix[i][i] = 0 // No vertex connects to itself
    for i = 0 to n-1 do
        for j = i+1 to n-1 do // Only iterate over upper triangle to avoid redundant
assignments
            matrix[i][j] = matrix[j][i] // If i is connected to j, j is connected to i
    return matrix
```

5. Algorithm Implementations

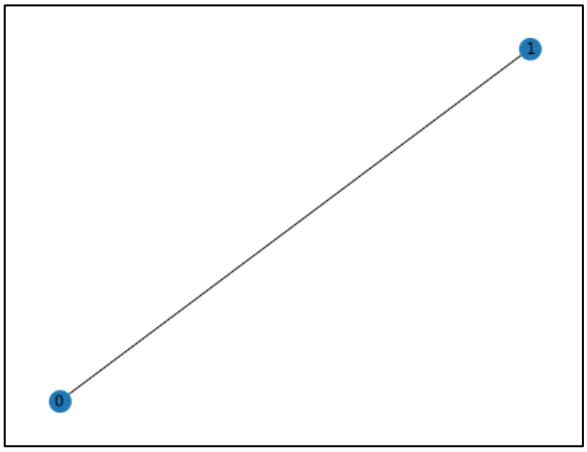
a. Brute Force Algorithm

Implementation of the brute force algorithm can be accessed via the following link: https://colab.research.google.com/drive/10R8Bq2zZ0oa16X2WEjILDNicO27uKJo1?usp=sharing

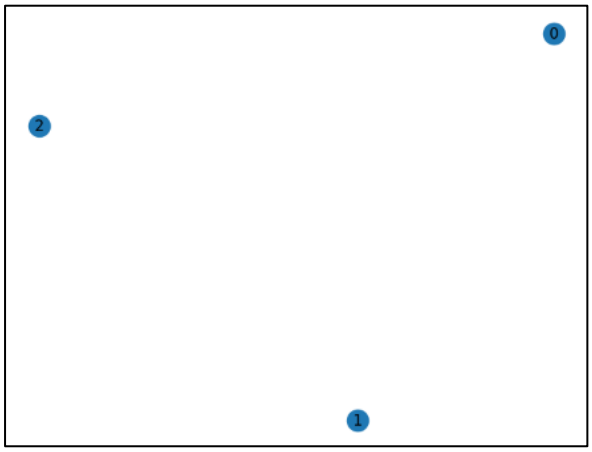
We performed the initial testing of the algorithm by using 15 samples.



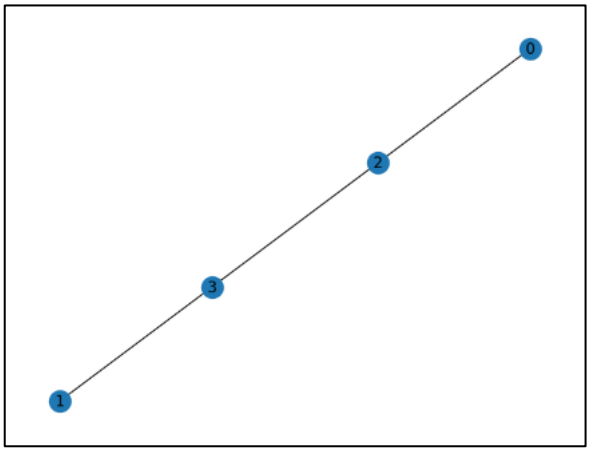
Sample 1: n = 1 | 1 x 1 matrix
Minimum Dominating Set: [0]



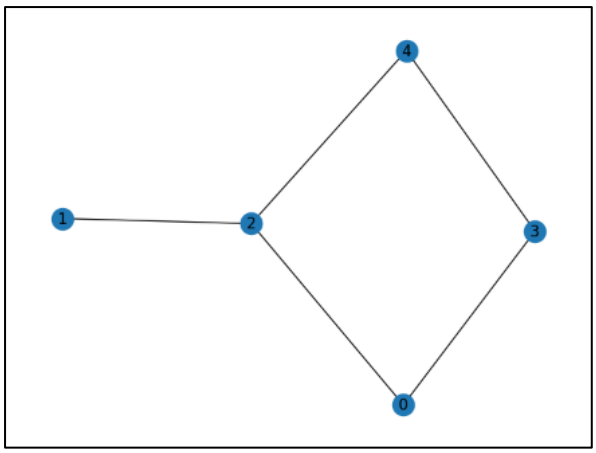
Sample 2: n = 2 | 2 x 2 matrix
Minimum Dominating Set: [0]



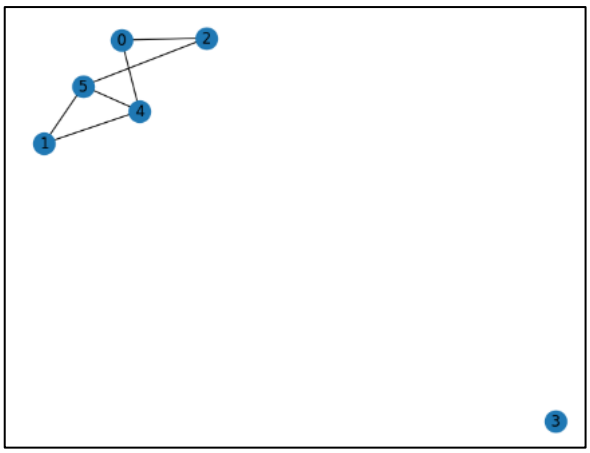
Sample 3: n = 3 | 3 x 3 matrix
Minimum Dominating Set: [0, 1, 2]



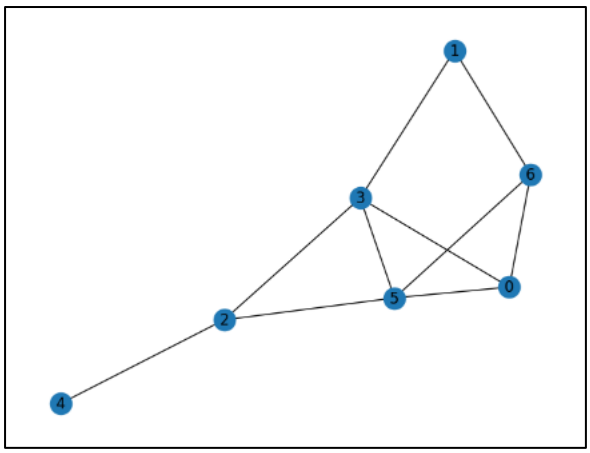
Sample 4: n = 4 | 4 x 4 matrix
Minimum Dominating Set: [0, 1]



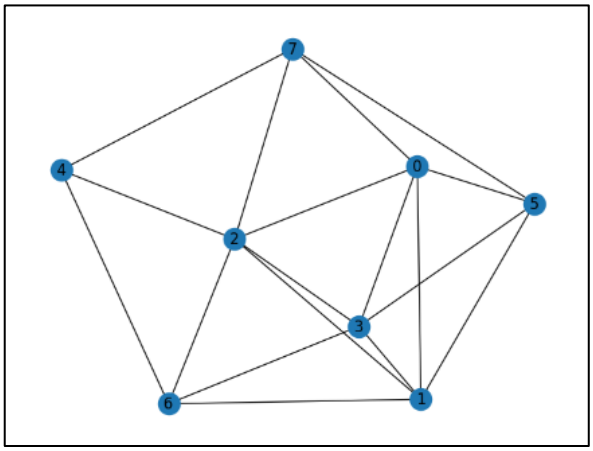
Sample 5: n = 5 | 5 x 5 matrix
Minimum Dominating Set: [0, 2]



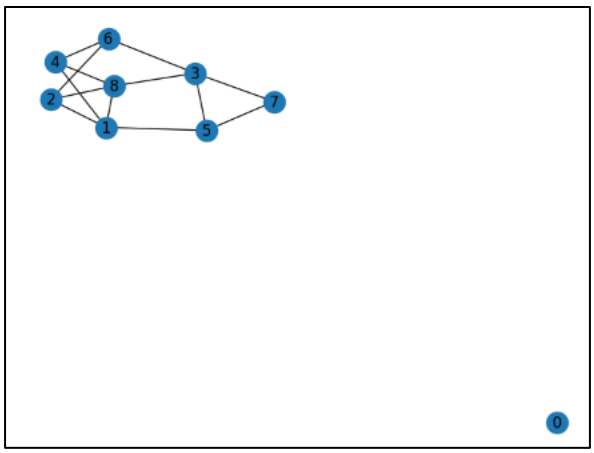
Sample 6: n = 6 | 6 x 6 matrix
Minimum Dominating Set: [0, 1, 3]



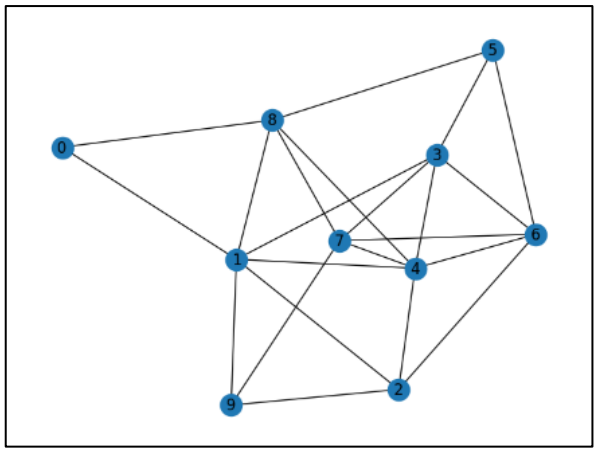
Sample 7: n = 7 | 7 x 7 matrix
Minimum Dominating Set: [0, 1, 2]



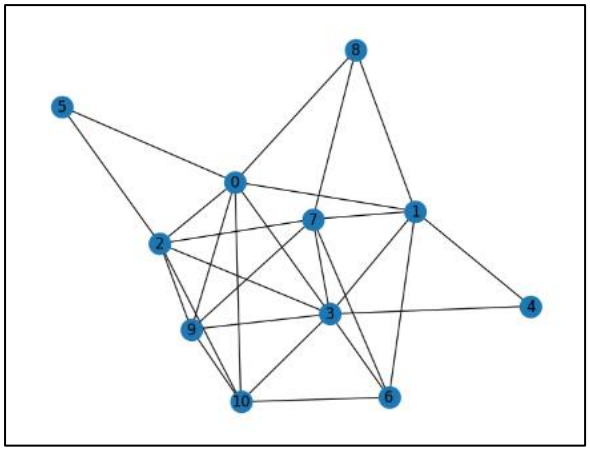
Sample 8: n = 8 | 8 x 8 matrix
Minimum Dominating Set: [0, 2]



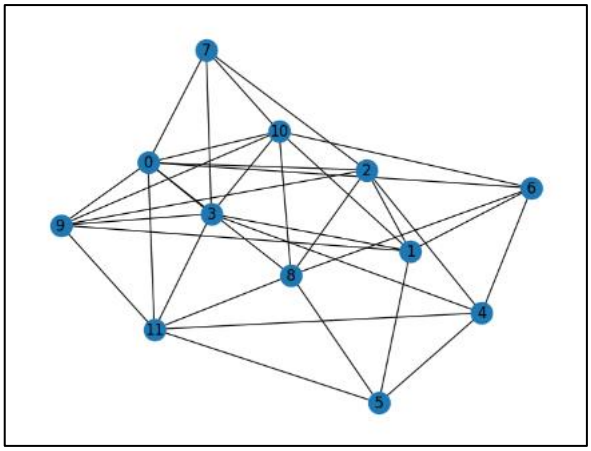
Sample 9: n = 9 | 9 x 9 matrix
Minimum Dominating Set: [0, 1, 3]



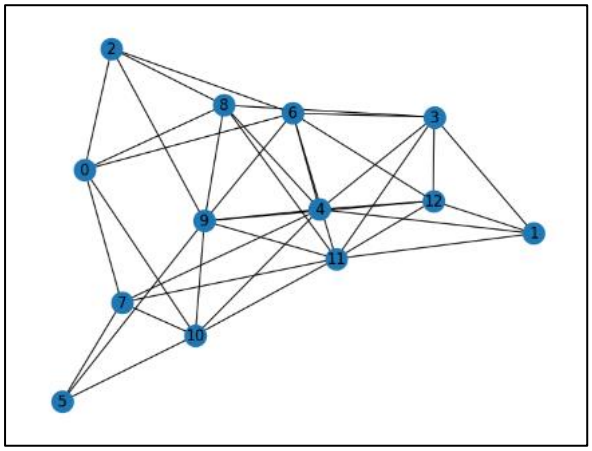
Sample 10: n = 10 | 10 x 10 matrix
Minimum Dominating Set: [1, 3]



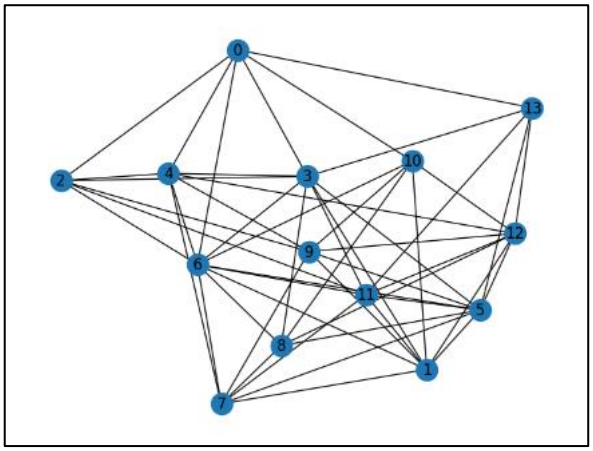
Sample 11: n = 11 | 11 x 11 matrix
Minimum Dominating Set: [0, 1]



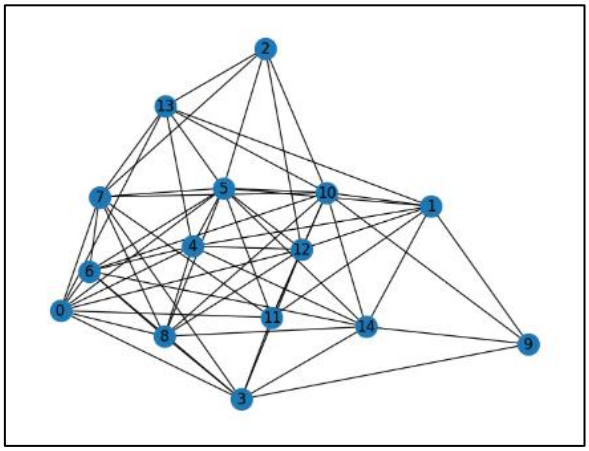
Sample 12: n = 12 | 12 x 12 matrix
Minimum Dominating Set: [0, 1, 2]



Sample 13: n = 13 | 13 x 13 matrix
Minimum Dominating Set: [0, 1, 5]



Sample 14: n = 14 | 14 x 14 matrix
Minimum Dominating Set: [1, 3]



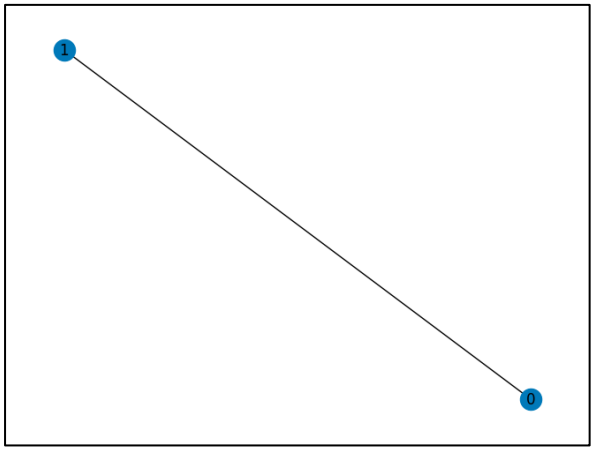
Sample 15: n = 15 | 15 x 15 matrix
Minimum Dominating Set: [0, 1, 2]

By analysing the samples provided above, it can be concluded that the algorithm worked properly on the 15 randomly generated instance. The important key point to emphasize is that the Minimum Dominating Set returned for each sample might not be the only solution (i.e., *Sample 4* also has a minimum dominating set of [2, 3]). The ones that the algorithm returns are having the minimum index values, this is resulting from the implementation design of the subset generation function. However, this does not cause any misleading solutions and further we can conclude that the algorithm works properly because of returning one of the minimum dominating sets possible as required.

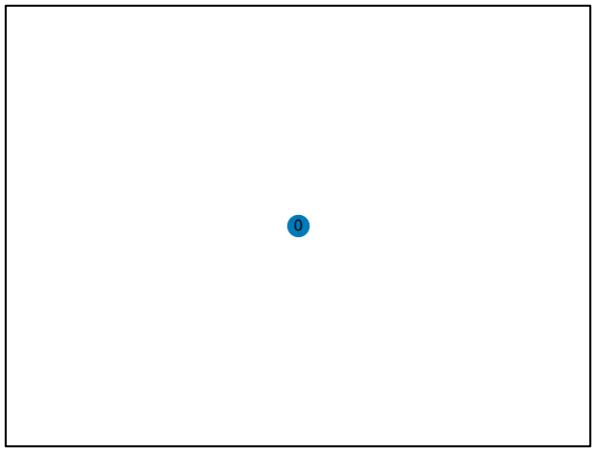
b. Heuristic Algorithm

Implementation of the approximation algorithm can be accessed via the following link: https://colab.research.google.com/drive/1Ofw7pnAaQC8tR-ACB38ShEck0cYy9Iex?usp=sharing

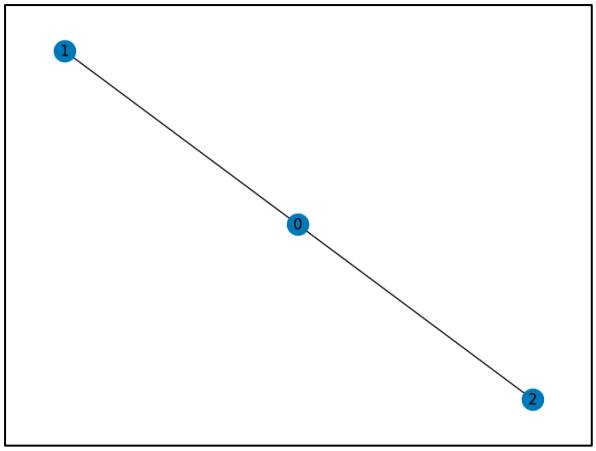
We performed the initial testing of the algorithm by using 15 samples.



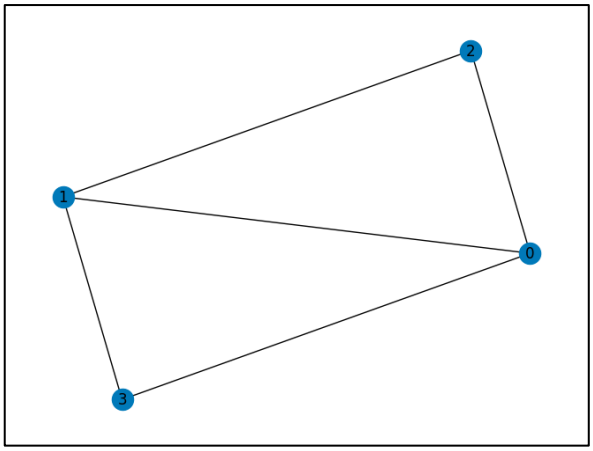
Sample 1: n = 1 | 1 x 1 matrix
Minimum Dominating Set: [0]



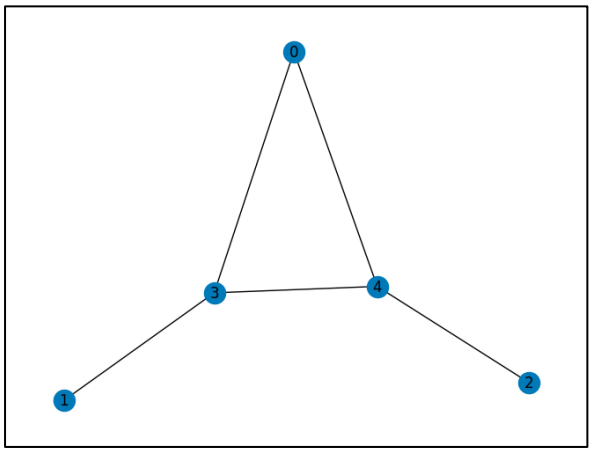
Sample 2: n = 2 | 2 x 2 matrix
Minimum Dominating Set: [0]



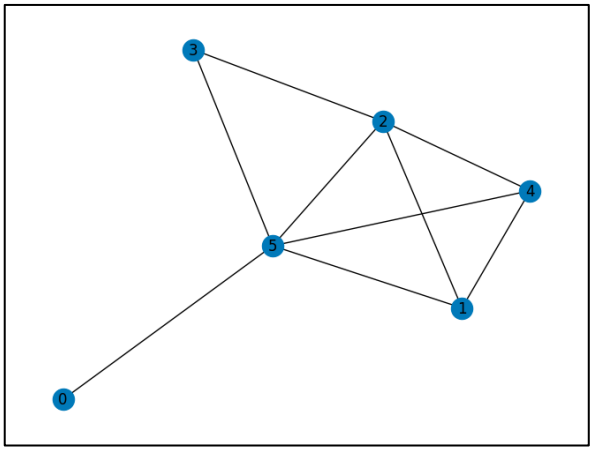
Sample 3: n = 3 | 3 x 3 matrix
Minimum Dominating Set: [0]



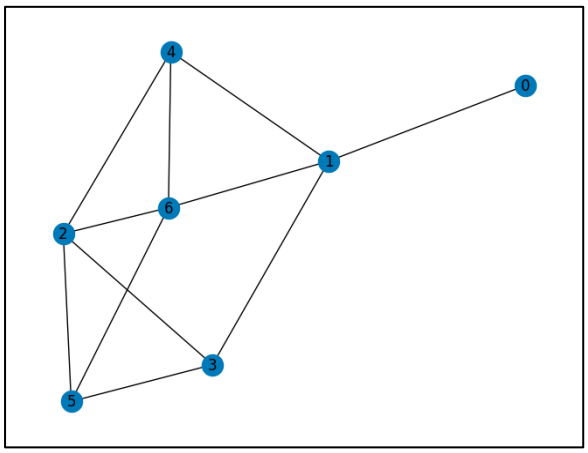
Sample 4: n = 4 | 4 x 4 matrix
Minimum Dominating Set: [0]



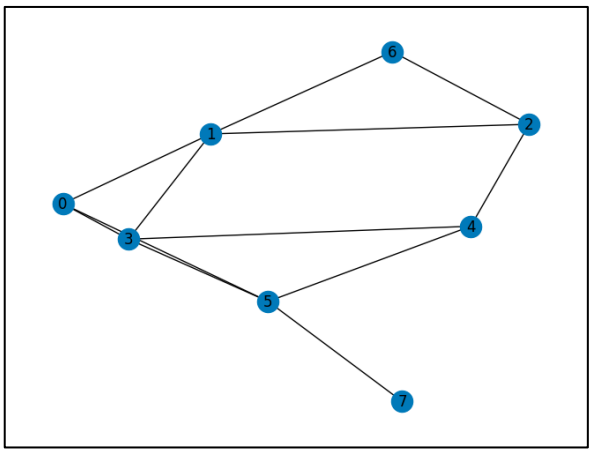
Sample 5: n = 5 | 5 x 5 matrix
Minimum Dominating Set: [2, 3]



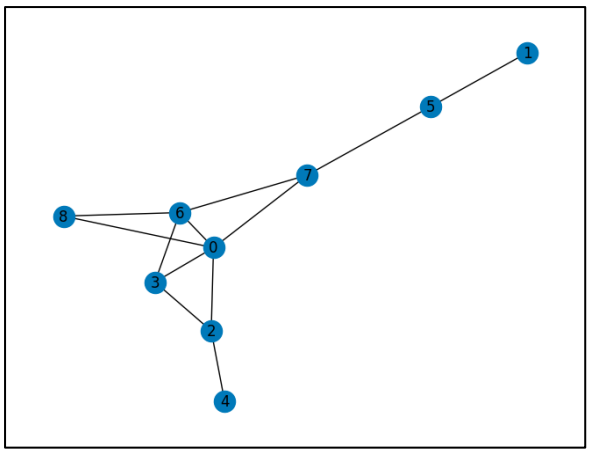
Sample 6: n = 6 | 6 x 6 matrix
Minimum Dominating Set: [5]



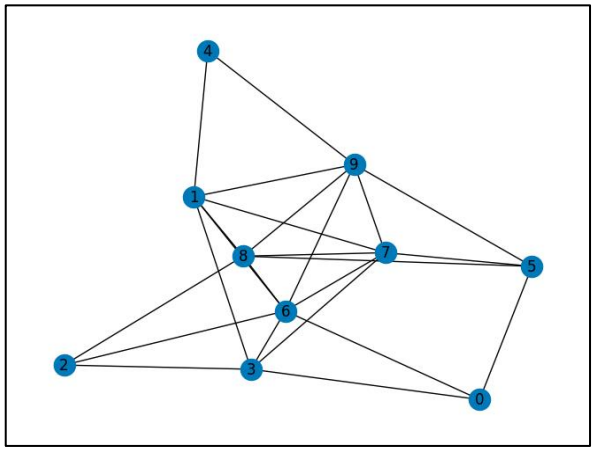
Sample 7: n = 7 | 7 x 7 matrix
Minimum Dominating Set: [1, 2]



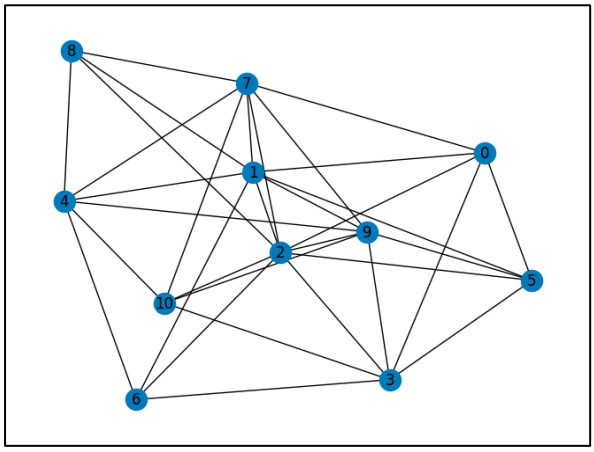
Sample 8: n = 8 | 8 x 8 matrix
Minimum Dominating Set: [1, 5]



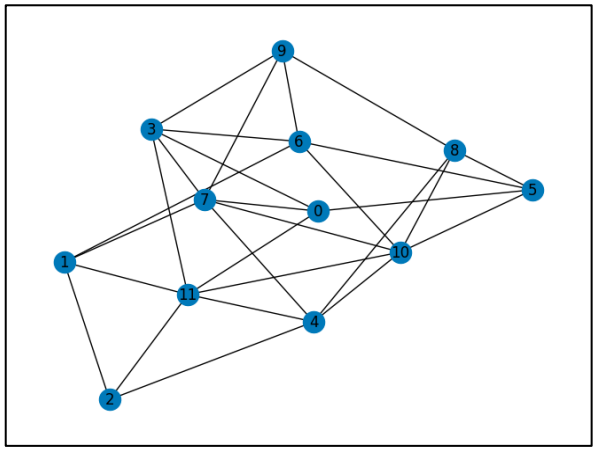
Sample 9: n = 9 | 9 x 9 matrix
Minimum Dominating Set: [0, 4, 5]



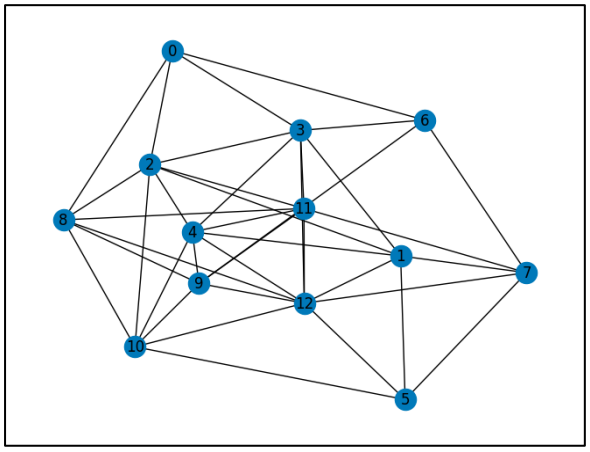
Sample 10: n = 10 | 10 x 10 matrix
Minimum Dominating Set: [4, 5, 6]



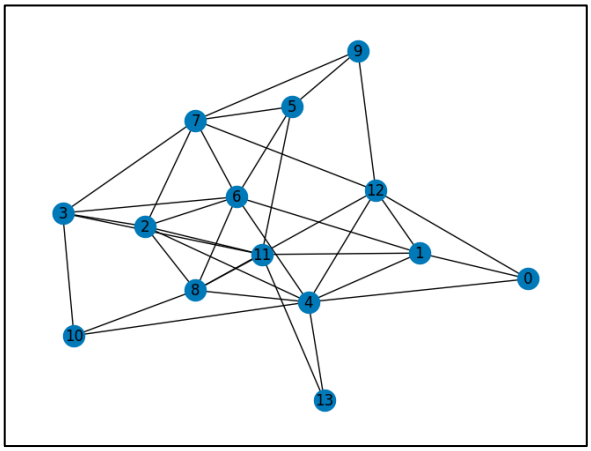
Sample 11: n = 11 | 11 x 11 matrix
Minimum Dominating Set: [2,4]



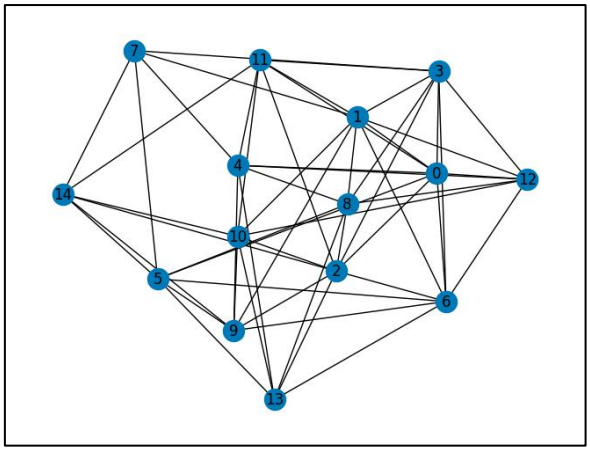
Sample 12: n = 12 | 12 x 12 matrix
Minimum Dominating Set: [8,11, 6, 7]



Sample 13: n = 13 | 13 x 13 matrix
Minimum Dominating Set: [2, 12, 6]



Sample 14: n = 14 | 14 x 14 matrix
Minimum Dominating Set: [11, 4, 7]



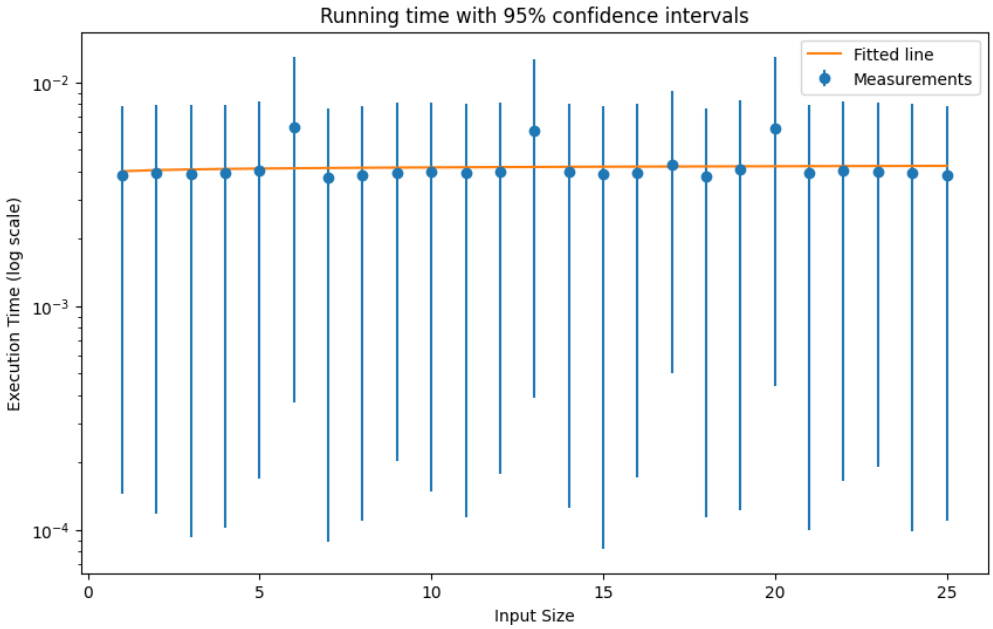
Sample 15: n = 15 | 15 x 15 matrix
Minimum Dominating Set: [1, 2, 4, 5]

6. Experimental Analysis of The Performance (Performance Testing)

In Performance testing we included graphs with nodes between 1 and 25. Number of repetitions for each input size is set to be 50 and analysed the mean run time of the heuristic algorithm. In order to analyse the outcomes of the test in a more precise way, we used both 90% and 95% confidence intervals. In other words, for an input size ranging from 1 to 25, we took 50 measurements for the running time for the confidence levels 90% and 95%. After analysing confidence intervals for each input size, we fit a line to our findings by using a log-log plot because our expected time complexity was V^2. After analysing the confidence intervals, we found out that the run time change we observe for increasing input size we do not observe a strict increase as we observed in Section 3-b, that is $O(V^2)$. Although $O(V^2)$ is an upper bound for out algorithm, it is not a tight bound. Probably it is because $O(V^2)$ is the worst-case scenario. Additionally, by fitting a line, we found out the following equation for our problem:

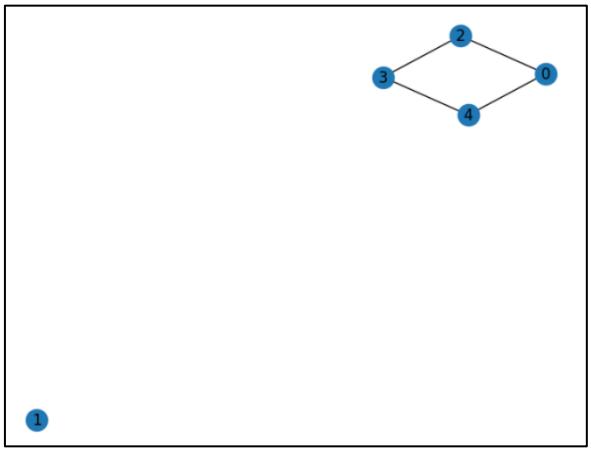
$$T(n) = 0.004013505707700045 * n^{0.017186036580502374}$$

Running time with 955 confidence intervals we have found is given below:

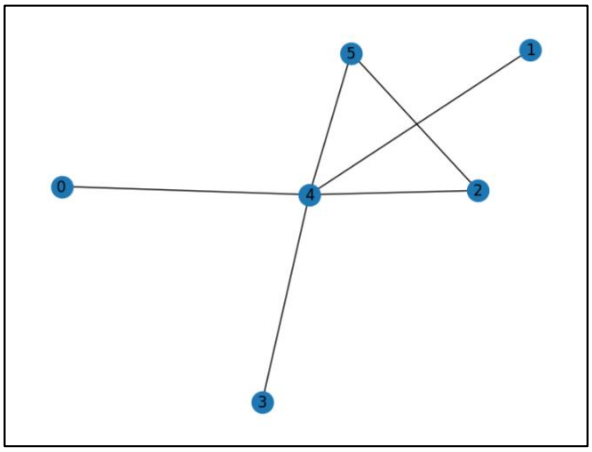


7. Experimental Analysis of the Quality

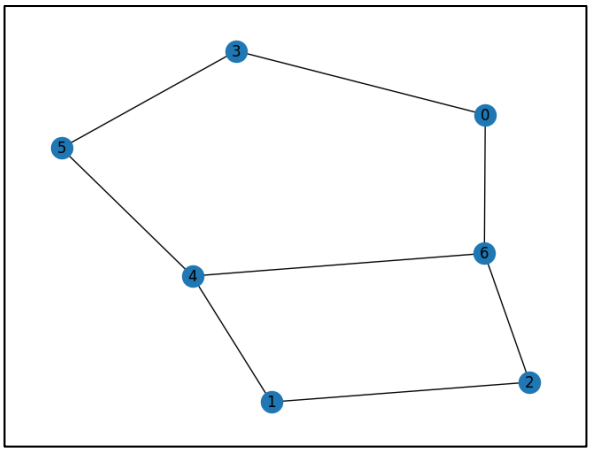
Although heuristic function is returning a minimum dominating set faster, it does not mean that it will not necessarily return the optimal solution. In order to understand the quality of heuristic algorithm we have used; we performed 10 experiments of different input sizes and compared the outputs of the heuristic and brute-force algorithm. Samples used in experiments and the outputs of the algorithms are shown below.



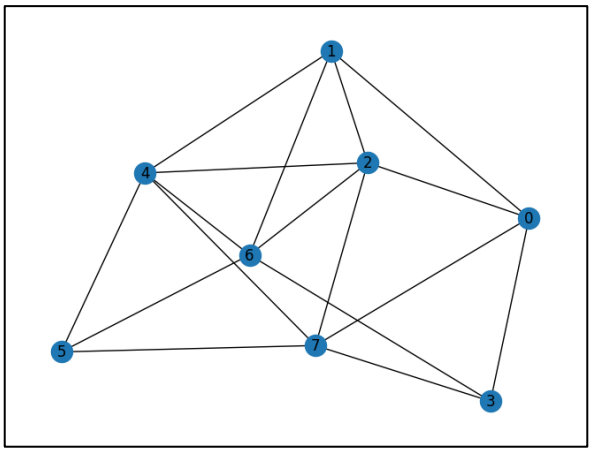
Sample 1: n = 5 | 5 x 5 matrix
MDS (Brute-Force): [0, 1, 2]
MDS (Heuristic): [0, 1, 3]



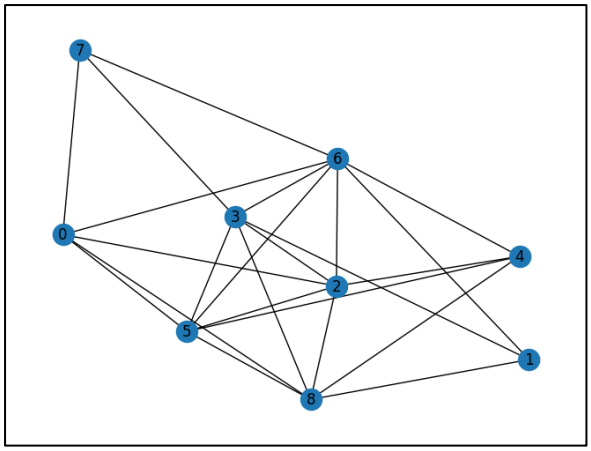
Sample 2: n = 6 | 6 x 6 matrix
MDS (Brute-Force): [4]
MDS (Heuristic): [4]



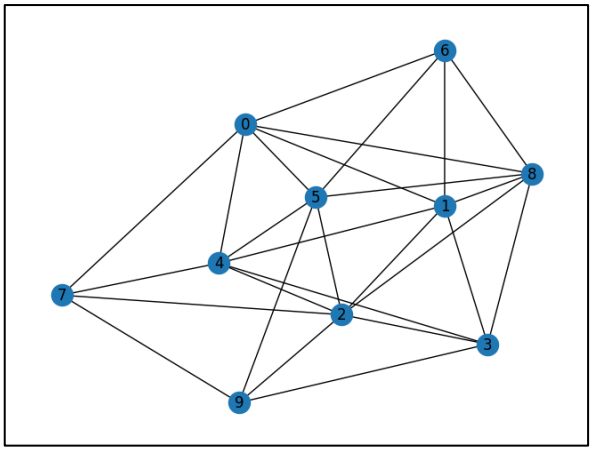
Sample 3: n = 7 | 7 x 7 matrix
MDS (Brute-Force): [0, 1, 3]
MDS (Heuristic): [0, 2, 4]



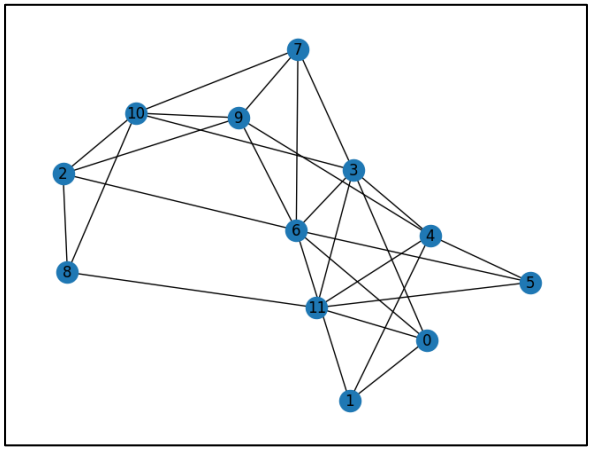
Sample 4: n = 8 | 8 x 8 matrix
MDS (Brute-Force): [0, 4]
MDS (Heuristic): [2, 3, 5]



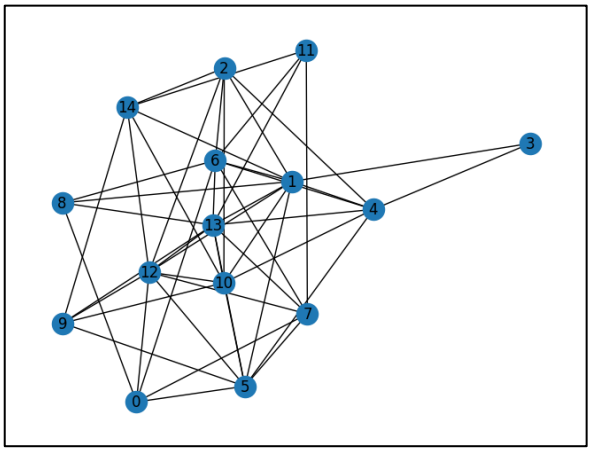
Sample 5: n = 9 | 9 x 9 matrix
MDS (Brute-Force): [2, 3]
MDS (Heuristic): [8, 6]



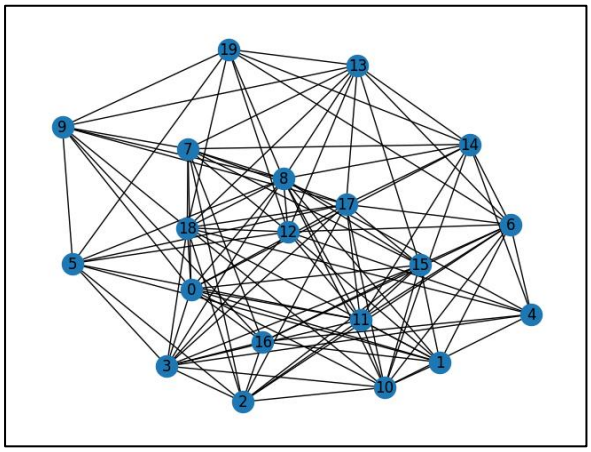
Sample 6: n = 10 | 10 x 10 matrix
MDS (Brute-Force): [0, 2]
MDS (Heuristic): [0, 2]



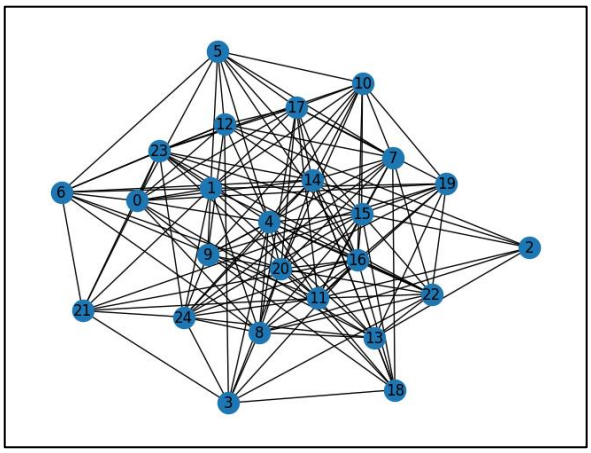
Sample 7: n = 12 | 12 x 12 matrix
MDS (Brute-Force): [2, 3, 4]
MDS (Heuristic): [10, 4, 6]



Sample 8: n = 15 | 15 x 15 matrix
MDS (Brute-Force): [1, 5, 6]
MDS (Heuristic): [1, 9, 7]



Sample 9: n = 20 | 20 x 20 matrix
MDS (Brute-Force): [2, 8]
MDS (Heuristic): [8, 17, 16]



Sample 10: n = 25 | 25 x 25 matrix
MDS (Brute-Force): [0, 7, 8]
MDS (Heuristic): [1, 2, 4, 7]

After the analysis performed, we observed probability of heuristic function to give the optimal solution decreases when the input size increases. When analysing the experiment result, it should be considered that the important think is the sizes of the minimum sets given by the algorithms because there can be several optimal solutions of the same sizes.

Heuristic algorithm gives correct result for samples 1, 2, 3, 5, 6, 7, 8; however, incorrect result for samples 4, 9, 10.

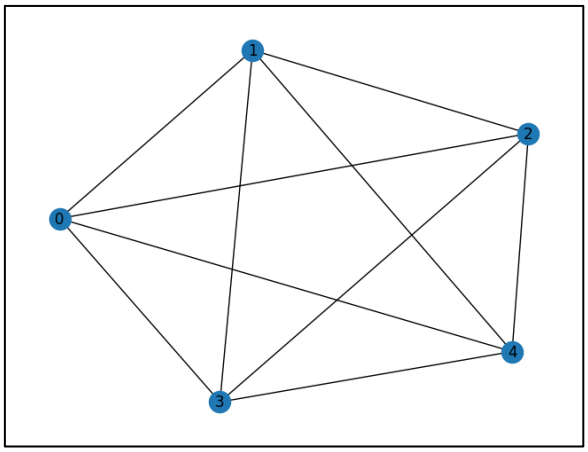
8. Experimental Analysis of the Correctness (Functional Testing)

The correctness results given in *Section 3-b* show that the algorithm is correct. However, there can be some errors resulted by the implementation of the algorithm, in other words codding errors. In order to test the functioning of the algorithm we performed both Black Box and White Box testing. Detailed implementation of the functional testing can be accessed via the Google Collab Notebook[1].

For Black Box testing, we performed experiments based on the extreme and/or edge cases that we have decided. The heuristic algorithm passed all of the test we introduced. Test cases are given below:

- **Test Case 1:** Empty adjacency matrix.
  - Expectation: The algorithm should return an empty set.
  - Result: Passed.
- **Test Case 2:** A graph with single vertex.
  - Expectation: The algorithm should return [0].
  - Result: Passed.
- **Test Case 3:** A graph with two disconnected vertices.
  - Expectation: The algorithm should return both vertices as MDS, [0, 1].
  - Result: Passed.
- **Test Case 4:** A complete graph consisting of 5 vertices.
  - Expectation: The algorithm should return a set of length 1, including just one of the nodes.
  - Result: Passed.

For White Box testing, we examined the statement and decision coverage of the algorithm. Performed testing with a test suite of one test case as give below:



Sample used in White Box Testing

Heuristic algorithm is having 100% statement coverage and 100% decision coverage.

[1] https://colab.research.google.com/drive/10R8Bq2zZ0oa16X2WEjILDNicO27uKJo1?usp=sharing

9. Discussion

In this study, the results of the algorithmic approach to the dominating set problem are examined and a detailed analysis of its performance is conducted. Moreover, the consistency between the theoretical analysis and experimental results are tested.

The approximation algorithm showcased promising results, displaying high accuracy and efficiency in solving the dominating set problem. The experimental analysis revealed that the approximation algorithm outperformed the brute force algorithm in terms of runtime, achieving significantly faster execution times. This improvement in efficiency can be crucial in real-world applications.

However, it is important to acknowledge that the brute force algorithm consistently yielded the correct results, albeit at a slower pace. On the other hand, our heuristic algorithm, did not always produce the optimal solution. Despite this drawback, it is worth noting that the heuristic algorithm demonstrated an approximation ratio of ($\ln \Delta + 2$), as proven in theoretical analysis. This approximation ratio indicates that the heuristic algorithm provided close enough results the dominating set problem.

Regarding the consistency between the theoretical and experimental analyses, some disparities were identified. After experimental analysis, we found out that the run time change we observe for increasing input size do not increase strictly as we observed in Section 3-b, that is $O(V^2)$. This inconsistency underscores the importance of refining the theoretical models to consider a broader range of factors and complexities present in real-world scenarios.

References

Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, NY: W. H. Freeman and Company.

Kuhn, F. (2013). Chapter 7: Dominating Sets. In Network Algorithms (pp. 63-65). University of Freiburg. https://ac.informatik.uni-freiburg.de/teaching/ss_13/netalg/lectures/chapter7.pdf