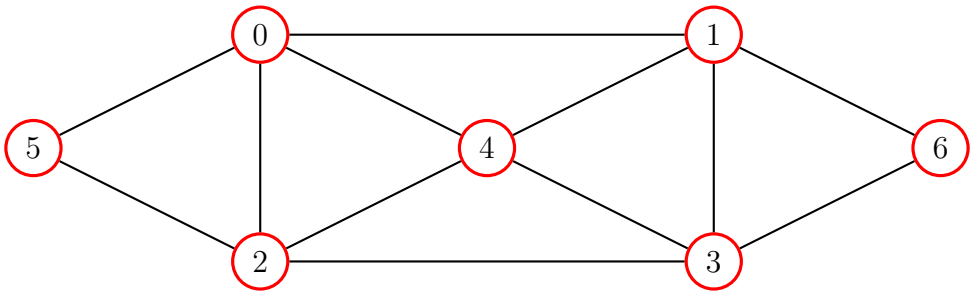
# Minimum Dominating Set of a Graph

Frank Bedekovich, Robert Krency, Regina Thase

November 9, 2021



# 1 Introduction

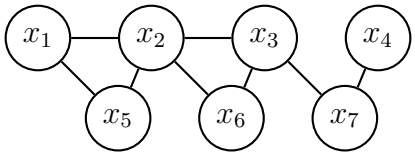
In this paper, we explore the Minimum Dominating Set of a Graph and how to find it. The Minimum Domianting Set of a Graph is the Set of all nodes such that every node or one of its neighbors is in the set.

This is in demonstration of the time complexity of algorithms, often denoted using Big-O notation ($O(n)$). To date, the best known algorithms used to guarantee the minimum dominating set are brute force approaches that run in exponential scales, of the form $O(2^n)$. Solutions however are able to be verified in polynomial time, of the form $O(n^2)$. It is with these two criteria that we classify the Minimum Dominating Set problem as an *NP-Complete* problem.

In this paper, we explore a brute force algorithm used to find the Minimum Domating Set. We also explore an approximate algorithm in order to compare the efficacy of the two, with accuracy and runtime being the major criteria.

# 2 Background

The Minimum Dominating Set of a Graph is defined as the set of vertices for which each vertex or one of its neighbors is in the set. The size of the minimum set will be unique, however there may be multiple sets of that size that are a minimal covering. Consider the following graph:




The vertex $x_2$ covers vertices $\{x_1, x_3, x_5, x_6\}$, leaving nodes $x_4$ and $x_7$. Either vertex can dominate the other and complete a covering of the graph. Thus, this graph has two minimum dominating sets, $\{x_2, x_4\}$ and $\{x_2, x_7\}$.

This problem is computationally difficult, and scales exponentially with the size of the graph. The next two sections cover the Brute Force approach and the Approximation approach, respectively.

# 3 Brute Force Approach

Calculating the Minimum Domating Set of a graph is a difficult problem. The size and complexity of a graph quickly grows with the number of nodes, $n$, and the number of edges, $e$, that make up the graph. As such, the only known way to guarantee finding the minimum dominating set is through a brute force approach.

This entails checking every possible combination of nodes in the graph. For each combination, the algorithm would check if it dominates every node in the graph, and if it is then checks if it's smaller than the currently known smallest dominating set. After all combinations are checked, the smallest known set is then guaranteed to be the Minimum Dominating Set.

To put this concept into pseudocode:

**Algorithm 1** Minimum Dominating Set brute force algorithm

```
1: procedure MINIMUM DOMINATING SET
2:     i ← 1
3:     while i < 2^n do
4:         Convert i into a binary array, binArray
5:         Create an array of 0s for the dominated nodes, domArray
6:         for each 1 ∈ binArray do
7:             Get the node at that index from the graph
8:             for each neighbor of the node do
9:                 Set it to dominated in domArray
10:            end for
11:        end for
12:        if all nodes dominated then
13:            if Current Solution Size < Best Solution Size then
14:                bestSolution ← i
15:            end if
16:        end if
17:        Increment i
18:    end while
19: end procedure
```

As this is a brute force algorithm, the amount of iterations needed is strictly dependent on the size of the graph. We can represent each solution as a set of bits, such that any index with a 1 indicates that node is a part of the dominating set. The number of possible solutions is therefore $2^n$, resulting in a runtime complexity of $O(2^n)$. The amount of iterations quickly scales up, crossing the one billion mark for a graph of size 30. This indicates that any sufficiently large graphs are essentially outside the limits of brute force computing. We later cover experimental data that shows this.

# 4 Approximation Approach

As the Minimum Domating Set requires a brute force approach, sufficiently large graphs can have enormous run times. To counter these problems, approximation algorithms are developed. These are designed to find an answer that is often considered good enough for the task at hand, while having a much lower complexity.

We explored one such approach. This approach uses a greedy technique for finding a dominating set. While the set is not covered, it selects the node with the highest out-degree of nodes that are not yet dominated. This node gets added to the current list of dominating nodes, and repeats until the entire graph is covered. This greatly improves our runtime complexity to $O(n^2)$.

To put this concept into pseudocode:

**Algorithm 2** Minimum Dominating Set approximation algorithm

```
1: procedure APPROXIMATE DOMINATING SET
2:     while the graph is not dominated do
3:         for each node not yet domianted do
4:             Find its out-degree of nodes not yet dominated
5:         end for
6:         Get the node with the highest out-degree
7:         Set all of its neighbors to be dominated
8:     end while
9: end procedure
```

As evident, this algorithm is much simpler. However, there is no guarantee that this algorithm will produce a minimal set, or even a set that is close to minimal. Depending on the task at hand, this approximate solution using a greedy technique may be preferred to a guaranteed minimal set.

We show some results of our experimental data in the next section.

# 5 Experimental Data

Utilizing the two algorithms outlined above, we set out to test them with some randomized, undirected graphs. We did this by generating random adjacency matrices. Each node had a connection to itself for ease of comparsion. Each node then had a chance to be connected to each other node, which we termed the graph density. We used a value of 30% for this graph density.

Then for each randomly generated graph, we ran both the brute force and approximation algorithms on it. We did starting with a graph of 20 nodes, and ran these algorithms on

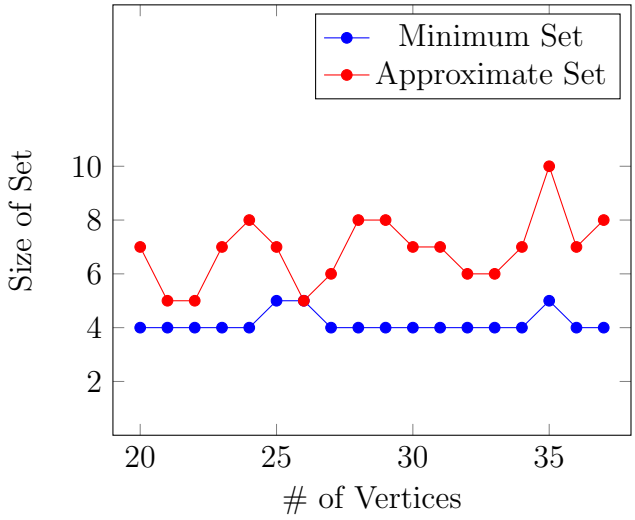
a new randomized graph of incrementally larger size until the runtime exceeded 24 hours. This happened at the 37-node mark with our implementation and hardware. The data is below:

| | Minimum Set | | Approximate Set | |
|---|---|---|---|---|
| **# of Vertices** | **Size** | **Time (seconds)** | **Size** | **Time (seconds)** |
| 20 | 4 | 0.46 | 7 | 8.6 e-6 |
| 21 | 4 | 1.00 | 5 | 6.5 e-6 |
| 22 | 4 | 2.09 | 5 | 6.6 e-6 |
| 23 | 4 | 4.46 | 7 | 9.2 e-6 |
| 24 | 4 | 9.44 | 8 | 1.0 e-5 |
| 25 | 5 | 20.15 | 7 | 9.6 e-6 |
| 26 | 5 | 42.07 | 5 | 8.9 e-6 |
| 27 | 4 | 90.59 | 6 | 1.0 e-5 |
| 28 | 4 | 233.23 | 8 | 1.2 e-5 |
| 29 | 4 | 395.63 | 8 | 1.2 e-5 |
| 30 | 4 | 852.31 | 7 | 1.3 e-5 |
| 31 | 4 | 1,834.03 | 7 | 1.4 e-5 |
| 32 | 4 | 3,869.44 | 6 | 1.4 e-5 |
| 33 | 4 | 8,145.86 | 6 | 1.4 e-5 |
| 34 | 4 | 17,353.50 | 7 | 1.5 e-5 |
| 35 | 5 | 35,943.60 | 10 | 1.9 e-5 |
| 36 | 4 | 75,798.40 | 7 | 1.8 e-5 |
| 37 | 4 | 161,439.00 | 8 | 2.0 e-5 |

The doubling of the time needed to run each incrementally larger graph through the brute force algorithm is easily spotted in the data. While a 37-node graph is not quite double the size of a 20-node graph, the time required ballooned from a half-second to nearly 45 hours. As the trend will continue due to the nature of the brute force approach, we can extrapolate that a 38-node graph will take approximately 90 hours to compute the minimum set. Further extrapolation reveals a 50-node graph will take over 42 *years*.

On the other hand, the approximation algorithm ran near instantaneously for each graph, with its growth much lower. The runtime of the approximation algorithm was measured in microseconds. As is apparent, the approximation algorithm runs in orders of magnitude faster time than the brute force approach.

The question remains though, just how efficient is the approximation algorithm in terms of the size of the minimum set? Below is a chart showing the relation between the minimum size and the approximate size for the data recorded above.




As this chart shows, the approximate solution only found a minimum solution one time. This indicates that it is entirely possible for the approximate solution to agree with the brute force approach. However, the chart also clearly shows that the majority of cases in our test data were much larger than the minimal solution, often times double the size.

# 6 Conclusion

This paper has explored two approaches to finding the Minimum Domating Set of a graph, a brute force approach and an approximation approach. This problem is an $NP-Complete$ problem. This follows from two facts about the problem. First, the best known solution is a brute force approach, checking every possible solution. Second, the solution is verifiable in polynomial time.

The brute force approach checks every possible combination of nodes of the graph to find a minimal covering. Thus, for a graph with $n$ nodes, the amount of possible combinations is $2^n$, indicating a runtime complexity of $O(2^n)$. Our experimental data shows that a modest 37-node graph can take nearly 45 hours to run. This does come with the guarantee of finding the minimal solution.

The approximation approach is much faster however. Utilizing a naive greedy approach, a 37-node graph was near instantaneous for a solution. For all experimental data, the

approximate solution ran orders of magnitude faster than the brute force approach. The implemented approach ran with a time complexity of $O(n^2)$, much better than the brute force approach. However, this comes with a major trade-off. The approximation algorithm rarely finds a minimal solution, with approximate solutions often being double the size of the found minimal solution.

Whether the brute force approach or the approximate approach is the proper choice is domain specific. Each project will thus need to weigh its factors in determining if an approximate solution is of sufficient quality when applying these techniques.