Sabancı University

Spring 2023

CS301 – Algorithms

Group 066 - Final Report

Dominating Set Problem

Selin Ceydeli, Canberk Tahıl

1. PROBLEM DESCRIPTION

The Dominating Set Problem (DSP) is an important problem in computer science concerning graph theory. The problem aims to find the minimum size dominating set of an undirected graph G. DSP has many practical applications and is widely used at subareas of computer networks including network design, social network analysis, and computer network security.

Intuitively, a dominating set is a subset of vertices S such that every vertex in the graph is either an element of the set S or adjacent to at least one element of the set S. The problem asks for the smallest size dominating set that satisfies this condition. Figure 1 below displays three dominating sets of the same graph:

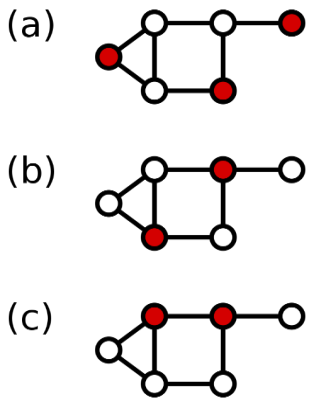


Figure 1. Three Dominating Sets of the Same Graph

The vertices of the dominating sets are colored in red. As it can be inferred from the graph, the minimum size dominating set of this graph has a cardinality 2. In other words, there is no dominating set with only 1 vertex.

Formally, the definition of the Dominating Set problem can be given as the following: Given an undirected graph $G = (V, E)$, a subset of vertices $D \subseteq V$ is called a dominating set if for every vertex $u \in V \setminus D$, there is a vertex $v \in D$ such that $(u,v) \in E$.[1]

The Minimum Dominating Set problem is a variant of the dominating set problem, where the goal is to find the smallest subset of vertices D that is a dominating set of the graph G. The formal definition of the Minimum Dominating Set problem is as follows:
Given an undirected graph $G = (V, E)$, a subset of vertices $D \subseteq V$ is called a dominating set if for every vertex $u \in V \setminus D$, there is a vertex $v \in D$ such that $(u,v) \in E$ and if there is no other subset of vertices D' that is a dominating set of G and has fewer vertices than D.[2]

The Minimum Dominating Set problem is NP-hard, which means that it is computationally difficult to solve the problem exactly in polynomial time, but it is possible to verify a solution at polynomial time. The Minimum Dominating Set problem is known to have approximately the same hardness as the Minimum Set Cover problem, which can be proved using standard reductions.[3] In the DS-SC reduction, every dominating set in graph G corresponds to a set cover of the same size in the set system (U, S), and vice versa. Hence, using results for Minimum Set Cover, we can obtain an $(H(\deg(G)+1) - 1/2)$-approximation algorithm for the Minimum Dominating Set problem.[4]

When we evaluate the implementation of reduction in the opposite direction, we can obtain results which are inapproximable for dominating set problems using two reductions. First,

---

[1] Casado, A., S. Bermudo, A.D. López-Sánchez, & Jesús Sánchez-Oro. "An iterated greedy algorithm for finding the minimum dominating set in graphs." Mathematics and Computers in Simulation 207 (2023): 41–58. doi: 10.1016/j.matcom.2022.12.018.
[2] Ibid.
[3] Hochbaum, D. S., Rossmanith, P., & Zangl, K. (2005). Approximation Algorithms for NP-hard Problems (pp. 201-241).
[4] Ibid.

the Split SC-DS reduction can be applied to an instance of Minimum Set Cover. This reduction produces a split graph, which is a graph whose vertex set can be partitioned into an independent set and a clique. Any set cover in (U, S) corresponds in the resulting split graph G to a dominating set (contained in S) of the same size (cardinality).[5] Second, the Bipartite SC-DS reduction can be applied to an instance of Minimum Set Cover. This reduction creates a bipartite graph with a bipartition (U, S), and two new vertices y and y' connected to each $S \in S$ and to each other. One can now confine to dominating sets consisting of y and a subset of S corresponding to a set cover, and hence we have $ds(G) = cds(G) = tds(G) = sc(G) + 1$.[6]

In order to adapt Feige's approximation lower bound from Minimum Set Cover to the lower bound for Minimum Dominating Set using split and bipartite SC-DS reductions, we need a hardness result on instances of set cover satisfying the following condition: $\ln(|U| + |S|) \approx \ln(|U|)$.[7] This is actually true analyzing Feige's construction. In this way, one can obtain the logarithmic lower bound for Minimum Dominating Set even in split and bipartite graphs. Hence, it can be concluded that Minimum Dominating Set Problem is as hard as Vertex Cover Problem which proves that Minimum Dominating Set Problem is a NP-hard problem.

2. ALGORITHM DESCRIPTION

2.1 Brute Force Algorithm

There is not an efficient algorithm that solves the Dominating Set problem in polynomial time since it is characterized as a NP-hard problem. Therefore, to tackle this problem, we designed a brute force algorithm, which enumerates all possible subsets of V: set of all vertices and returns the subset with the smallest cardinality that satisfies the condition for being a dominating set. The step-by-step explanation of the algorithm is as follows:

---

[5] Chlebík, M., & Chlebíková, J. (2008). Approximation hardness of dominating set problems in bounded degree graphs. *Information & Combination,* (11), 1264–1275.

[6] Ibid.

[7] Uriel Feige, "On the Hardness of Approximating Minimum Vertex Cover" (1998), Journal of the ACM, vol. 45, no. 2, pp. 433-438.

1 Define two variables, smallest_set and smallest_cardinality, for respectively storing the smallest-cardinality set and for storing the smallest size of the dominating set.

2 Enumerate all subsets of the set V in an iterative way.

2.1 For each subset W generated, check if it is a dominating set by testing if each vertex in V-W has at least one neighboring vertex in W.

2.2 If the current subset W is a dominating set and has a smaller cardinality than the set stored in the smallest_set variable, update the smallest_set and smallest_cardinality variables accordingly.

3 Return the smallest set.

The pseudocode of the brute force algorithm is as follows:

```
procedure Minimum_Dominating_Set(G):
    initialize smallest_set to an empty set
    initialize smallest_cardinality to INT_MAX
    set n as length(G)
    for every i in range 2^n:
        initialize subset to an empty set
        for every j in range n:
            if ( i and (1 << j) ) not equal to zero: #Bit manipulation method for subset generation
                add the vertex G[j] to the subset
        if Is_Dominating(G, subset) is True and length(subset) < smallest_cardinality :
            update smallest_set to subset
            update smallest_cardinality to length(subset)
    return smallest_set

procedure Is_Dominating(G, W):
    for every vertex in G:
        if vertex is not in W:
            initialize has_neighbor to False
            for every neighbor of vertex in graph G:
                if neighbor is in W:
                    set has_neighbor to True
                    break from the inner for loop
            if has_neighbor is False:
                return False
    return True
```

Is_Dominating(G, W) function checks if a given set of vertices W is a dominating set for a graph G.

Minimum_Dominating_Set(G) function enumerates all possible subsets of vertices of the graph using bit manipulation and keeps track of the smallest subset found and returns it as the minimum dominating set of the graph.

Minimum_Dominating_Set(G) algorithm is an exponential-time algorithm, whose detailed worst-case time complexity analysis is made in Section 3: Algorithm Analysis. Since the efficiency of an algorithm is typically evaluated by its worst-case time complexity, it can be concluded that the Minimum_Dominating_Set(G), which solves the dominating set problem using a brute force approach, is not an efficient algorithm.

Moreover, the Minimum_Dominating_Set(G) algorithm can be regarded as an iterative algorithm because it uses a nested for loop to iterate over all possible subsets of vertices of the graph G using the bit manipulation technique and updates the smallest set found so far only when it finds a dominating set of smaller cardinality than the previous smallest dominating set found.

2.2 Heuristic Algorithm

Heuristic algorithms are used to attack NP-hard problems. Even though these algorithms do not give a guarantee of finding a correct answer, they work fast. With the heuristic algorithms, optimality and correctness of the solutions are traded for speed.

Since the Dominating Set problem is an NP-hard problem, a heuristic algorithm can be designed to attack this problem faster than the brute force approach we explained in the previous section. In this section, we will explain the greedy heuristic algorithm designed by Hernández Mira, Parra Inza, Sigarreta Almira, and Vakhania.[8]

[8] F. Á. Hernández Mira et al., "A Polynomial-Time Approximation to a Minimum Dominating Set in a Graph," Theoretical Computer Science, vol. 930, pp. 142-156, 2022, https://doi.org/10.1016/j.tcs.2022.07.020.

The pseudocode of the greedy heuristic algorithm is as follows:

procedure Greedy_Algorithm(G):
    initialize iteration_num to 0
    initialize v_0 to any vertex with the maximum degree in G
    $S_1 = \{v_0\}$
    while S_h is not a dominating set of G :
        increment iteration_num by 1
        initialize v_h to any vertex with the maximum active degree in set S'_{h-1}
        update S_h as the union of S_{h-1} and v_h
    return S_h

Above-defined greedy heuristic algorithm works on a number of iterations, denoted by iteration_num in the pseudocode. S_h represents the dominating set formed after iteration h. The selection of the vertex v_h in the while loop is based on the active degree of the vertices in set S'_{h-1}, where S'_{h-1} is the complement of the set S_{h-1}. The active degree of a vertex is calculated by subtracting the number of vertices already belonging to set S_{h-1} and vertices adjacent to a vertex in set S_{h-1} from the maximum degree of the vertex.[9]

At every iteration, the updated set S_h contains one more vertex than the set S_{h-1} of the previous iteration and thus, S_h enlarges at every iteration. The algorithm ends when S_h is already a dominating set.[10]

This heuristic algorithm operates on a greedy principle. It makes the seemingly optimal choice at every iteration by selecting the vertex with the highest active degree in the S_h-1 set. However, this algorithm does not necessarily yield the optimal solution since the algorithm chooses the seemingly optimal solution at each step without considering the globally optimal solution.

To discuss how well this greedy heuristic algorithm performs, a ratio bound analysis will be conducted. Firstly, the relationship between the cardinality of optimal dominating set

[9] F. Á. Hernández Mira et al., "A Polynomial-Time Approximation to a Minimum Dominating Set in a Graph," Theoretical Computer Science, vol. 930, pp. 142-156, 2022, https://doi.org/10.1016/j.tcs.2022.07.020.
[10] Ibid.

solution S* and the number of vertices in a connected, undirected graph G must be found. For a connected, undirected graph G with n vertices, an optimal dominating set can be constructed with at most n/2 vertices because each vertex can cover itself and at least one other vertex that is neighbor to it. Therefore, |S*| <= n/2.

Now, a relationship between the cardinality of the domanting set solution produced by the heuristic algorithm, |S|, and the number of vertices, n, must be found to construct the ratio bound.

In the worst-case, the greedy algorithm could add all n vertices to the dominating set S if each vertex is only connected to one other vertex that has not already been dominated. Therefore, |S| <= n.

The inequalities |S*| <= n/2 and |S| <= n do not explicitly imply that |S| <= 2|S*|, or equivalently |S|/|S*|<=2. Thus, a constant ratio bound ρ(n) cannot be derived for the greedy heuristic algorithm for the minimum dominating set problem. The lack of a ratio bound ρ(n) =2 will also be shown in Section 7: Experimental Analysis of the Quality.

3. ALGORITHM ANALYSIS

3.1 Correctness Analysis of the Brute Force Algorithm

Theorem: The Minimum_Dominating_Set(G) algorithm correctly finds the minimum dominating set of an undirected graph G.

Proof: Let D* be the minimum dominating set of G, and let D be the set returned by the Minimum_Dominating_Set(G) algorithm. We need to show that D = D*.

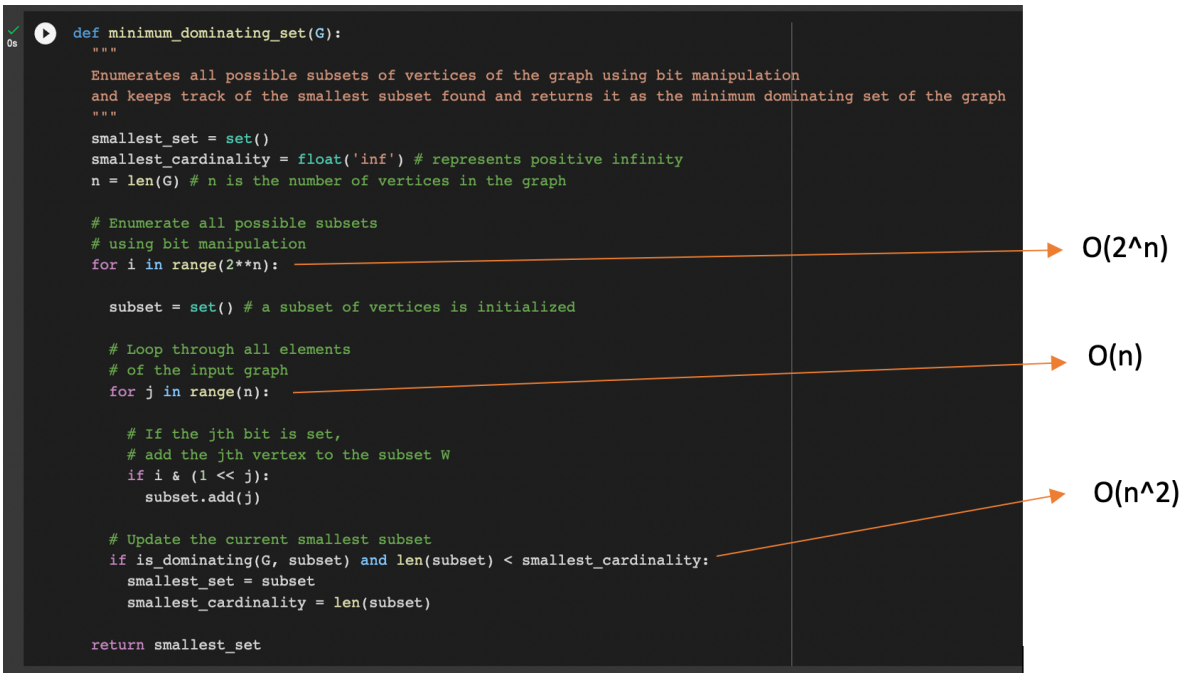
Before proving that D is a minimum dominating set of G, we first need to show that it is a dominating set of G. Since Is_Dominating(G, W) function returns true iff W is a dominating set of G, D can be said to be a dominating set of G if Is_Dominating(G, D) returns true. Next, we need to show that D is a minimum dominating set of G.

Suppose that D is a minimum dominating set of G. We will prove this statement by using the proof of contradiction. In order to cause a contradiction, we also assume that there exists another dominating set D' of G with a strictly smaller cardinality (i.e. |D'| < |D|). Since the Minimum_Dominating_Set(G) algorithm considers all possible subsets of vertices of G in non-decreasing order of cardinality, it must have considered D' before D. We know that D' is a dominating set of G because Is_Dominating(G, D') returns true.

The proof proceeds with the observation that the Minimum_Dominating_Set(G) algorithm updates the smallest set found so far only when it finds a dominating set of smaller cardinality than the previous smallest dominating set found. Thus, if Is_Dominating(G, D') returns true, and if |D'| < |D|, then D' must have been found by the algorithm before D and the smallest set cannot be updated to D in the upcoming iterations. However, this contradicts the assumption that D is the minimum dominating set of G.

Hence, it can be concluded that the assumption that there exists another dominating set D' of G with |D'| < |D| leads to a contradiction. Therefore, no such D' exists, and D is indeed a minimum dominating set of G.

3.2 Complexity Analysis of the Brute Force Algorithm



The worst-case time complexity of the Minimum_Dominating_Set(G) algorithm is O((2^n)*(n^2)) where n is the number of vertices in the graph G. The O(2^n) component of the complexity comes from the enumeration of all 2^n possible subsets of the vertices of the graph G and considering each one of them as a candidate for the minimum dominating set of graph G. The O(n^2) term in the complexity comes from the Is_Dominating(G, W) function, which checks whether each subset W is a dominating set by examining each vertex and its neighbors, which takes O(n^2) time in the worst case.

Thus, the upper bound for the worst-case time complexity of the brute force algorithm, i.e. the Minimum_Dominating_Set(G) algorithm, is shown to be O((2^n)*(n^2)).

3.3 Correctness Analysis of the Heuristic Algorithm

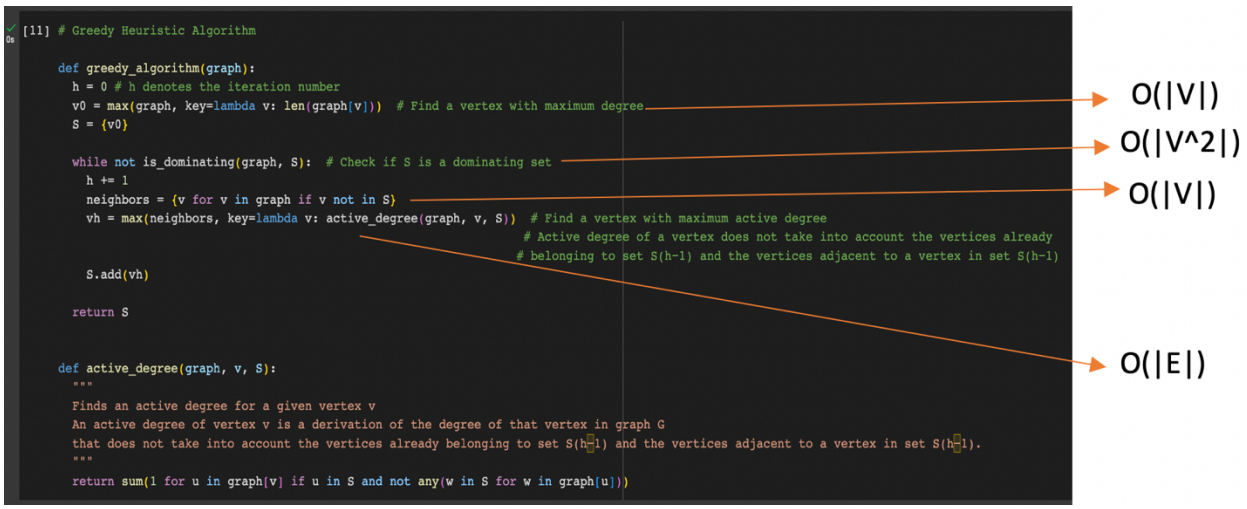
Theorem: If $|S(hmax)| \leq 2$, then S(hmax) is a minimum dominating set.

Proof: To prove this theorem, we need to prove the theorem for both cases: $|S(hmax)| = 1$ and $|S(hmax)| = 2$. In other words, we need to show that if $|S(hmax)| = 1$ or $|S(hmax)| = 2$, then S(hmax) is a minimum dominating set by deriving it from the heuristic algorithm.

For $|S(hmax)| = 1$, suppose there exists a $S^*=\{v0\}$ where S^* is the minimum dominating set. Then $v0 \in V$ where v0 is adjacent to every vertex in V and the outgoing degree of v0 is |V|-1. Before the iteration of the while loop in the greedy heuristic algorithm commences, the vertex with the highest outgoing edge cardinality is selected to be included in S_h which is initially an empty set. In an undirected acyclic graph, a vertex can have a maximum of |V|-1 many edges formed with different vertices. Since at the beginning of the proof, we assumed that v0 has an outgoing degree of |V|-1, v0 will be selected as the vertex to be included at S_h and S_h becomes {v0}. The algorithm checks whether S_h is a dominating set or not and evaluates to true because the cardinality of S_h is already 1. Since $|S^*| = 1$ and $|S_h| = 1$, it can be concluded that S_h is in fact the minimum dominating set with cardinality 1.

For |S(hmax)| = 2, suppose that there exists a S*={v0, v1} such that v0, v1 ∈ V are vertices which are adjacent to every other vertex in V. Let S_h be an empty set which will converge into a minimum dominating set as the algorithm terminates. Before the while loop in the greedy heuristic algorithm commences, the vertex with the highest outgoing edge cardinality is selected to be included in S_h, which is v0. Hence, S_h = {v0} and $|S_h|$ = 1. At the first iteration of the while loop, the vertex vh with the maximum active degree in set S'(h-1) will be selected. Since we assumed that v0 and v1 are adjacent to every other vertex in V, that is among all vertices v0 and v1 have the maximum active degrees. Since v0 has already been selected, vh will be selected as v1 and will be added to the set S_h. Hence, the updated version of S_h is: S_h = {v0, v1} and $|S_h|$ = 2. At the final iteration of algorithm, the algorithm checks if S_h is a dominating set or not and evaluates to true because the cardinality of S_h is already 2. Since S* = S_h = {v0, v1} and |S*| = $|S_h|$ = 2, it can be concluded that S_h is in fact the minimum dominating set with cardinality 2.

3.4 Complexity Analysis of the Heuristic Algorithm



The worst-case time complexity of the Greedy_Algorithm(G) is O(|V| + |V|^2 * (|V|+|E|)), which is approximately equal to O(|E|*|V|^2) where |V| is the number of vertices and |E| is the number of edges in the graph G. If the graph is sparse, that is |E| = |V|, then the complexity becomes O(|E|*|V|^2) = O(|V|^3), however if the graph is dense, that is |E| = |V|^2, then the complexity becomes O(|E|*|V|^2) = O(|V|^4).

The O(|V|) component of the complexity comes from the selection of the vertex with the maximum degree. As explained in Section 3.2: Complexity Analysis of the Brute Force Algorithm, Is_Dominating(G, S) function, which checks whether a subset S is a dominating set by examining each vertex and its neighbors, has a running time of O(|V|^2). At the inner loop, getting neighbors takes O(|V|) time since we iterate over all vertices in graph and it takes constant time to check whether vertex v is in set S. The active_degree(G, v, S) takes O(|E|) time since we are checking the number of its neighbors that are in S and do not have any neighbors in S.

Thus, the upper bound for the worst-case time complexity of the heuristic algorithm, i.e. the Greedy_Algorithm(G), is shown to be O(|E|*|V|^2).

4. SAMPLE GENERATION (SAMPLE INSTANCE GENERATOR)

Random instances are needed to test the performance of the algorithms and since the main topic of this report is the Dominating Set problem, the random instance generator will generate random undirected graphs. Dictionaries are used to represent graphs with their corresponding vertices and edges. The steps below show the creation of a random undirected graph with N vertices and an edge probability P:

1 The graph is represented as a dictionary with vertices as keys and sets of adjacent vertices as values.

1.1 For each vertex i, an empty set of adjacent vertices is created and initialized.

1.2 For each pair of distinct vertices i and j, generates a random number between 0 and 1.

1.2.1 If the generated number is less than P value, then the vertices i and j are connected by adding each other to their sets of adjacent vertices.

1.2.2 Repeat this process for all pairs of vertices.

2 Return the generated graph.

The pseudocode of the sample generator algorithm is as follows:

```
procedure Generate_Random_Graph(N, P):
    initialize graph to an empty dictionary
    for i from 0 to n-1:
    initialize an empty set of adjacent vertices for vertex i
    for i from 0 to n-1:
      for j from i+1 to n-1:
```

if i and j are distinct and a random number between 0 and 1 is less than P:
add j as an adjacent vertex for i
add I as an adjacent vertex for j
return graph

5. ALGORITHM IMPLEMENTATIONS

5.1 The Implementation of the Brute Force Algorithm

We implemented our brute force algorithm in Python and tested our implementation using 15 samples that we generated using our sample generator tool of Section 4.

```python
def minimum_dominating_set(G):
    """
    Enumerates all possible subsets of vertices of the graph using bit manipulation
    and keeps track of the smallest subset found and returns it as the minimum dominating set of the graph
    """
    smallest_set = set()
    smallest_cardinality = float('inf') # represents positive infinity
    n = len(G) # n is the number of vertices in the graph

    # Enumerate all possible subsets
    # using bit manipulation
    for i in range(2**n):

        subset = set() # a subset of vertices is initialized

        # Loop through all elements
        # of the input graph
        for j in range(n):

            # If the jth bit is set,
            # add the jth vertex to the subset W
            if i & (1 << j):
                subset.add(j)

        # Update the current smallest subset
        if is_dominating(G, subset) and len(subset) < smallest_cardinality:
            smallest_set = subset
            smallest_cardinality = len(subset)

    return smallest_set
```

```python
# A dominating set in a graph is a set of vertices such that
# every vertex outside the set is adjacent to a vertex in the set
def is_dominating(G, W):
  """
  Checks if a given set of vertices W is a dominating set for a graph G
  """
  for v in G:
    # Check every vertex outside the set
    # if it is adjacent to a vertex in the set
    if v not in W:
      is_neighbor_in_W = False
      for u in G[v]:
        if u in W:
          is_neighbor_in_W = True
          break
      if not is_neighbor_in_W:
        return False
  return True
```

```python
import random

nodes=[5, 10, 15, 20, 25]
prob=[0.25, 0.5, 0.75]

def generate_random_graph(n, p):
  """
  Generates a random undirected graph with n vertices and an edge probability p
  Returns a dictionary with vertices as keys and adjacent vertices as values
  """
  graph = {i: set() for i in range(n)}  # Initialize graph with all vertices and no edges
  for i in range(n):
    for j in range(i+1, n):  # Only iterate over j > i to avoid adding duplicate edges
      if random.random() < p:
        graph[i].add(j)
        graph[j].add(i)  # Add edge in both directions
  return graph

graphs=[]

for node in nodes:
  for p in prob:
    graphs.append(generate_random_graph(node,p))

for g in graphs:
  print(g)
```

```python
min_set = set()
min_set = minimum_dominating_set(graphs[0])
print(min_set)
```

```
{0, 4, 7}
```

5.2 The Results of the Initial Testing of the Brute Force Algorithm

The results of the initial testing of the brute force algorithm using 15 sample inputs are summarized in Table 1 below. D denotes the minimum dominating set generated by the algorithm and |D| denotes its cardinality.

Table 1. Results of the Initial Testing of the Brute Force Algorithm

		Number of Vertices at Graph (n)				
		N=5	N=10	N=15	N=20	N=25
Edge Probability (p)	p=0.25	D={1,2} |D|= 2	D={0,1,2,8} |D|= 4	D={0,1,6,7,10} |D|= 5	D={0,4,5,6,7} |D|= 5	D={2,10,21,23} |D|= 4
	p=0.5	D={0,3} |D|= 2	D={1,2} |D|= 2	D={4,7} |D|= 2	D={12,15} |D|= 2	D={2,4,10} |D|= 3
	p=0.75	D={0,1} |D|= 2	D={0} |D|= 1	D={0,1} |D|= 2	D={0,1} |D|= 2	D={3,4} |D|= 2

5.3 The Implementation of the Heuristic Algorithm

We implemented our heuristic in Python and tested our implementation using 15 samples that we generated using our sample generator tool of Section 4.

```python
# A dominating set in a graph is a set of vertices such that
# every vertex outside the set is adjacent to a vertex in the set
def is_dominating(G, W):
  """
  Checks if a given set of vertices W is a dominating set for a graph G
  """
  for v in G:
    # Check every vertex outside the set
    # if it is adjacent to a vertex in the set
    if v not in W:
      is_neighbor_in_W = False
      for u in G[v]:
        if u in W:
          is_neighbor_in_W = True
          break
      if not is_neighbor_in_W:
        return False
  return True
```

```
[11] # Greedy Heuristic Algorithm

     def greedy_algorithm(graph):
       h = 0 # h denotes the iteration number
       v0 = max(graph, key=lambda v: len(graph[v]))  # Find a vertex with maximum degree
       S = {v0}

       while not is_dominating(graph, S):  # Check if S is a dominating set
         h += 1
         neighbors = {v for v in graph if v not in S}
         vh = max(neighbors, key=lambda v: active_degree(graph, v, S))  # Find a vertex with maximum active degree
                                                                        # Active degree of a vertex does not take into account the vertices already
                                                                        # belonging to set S(h-1) and the vertices adjacent to a vertex in set S(h-1)
         S.add(vh)

       return S


     def active_degree(graph, v, S):
       """
       Finds an active degree for a given vertex v
       An active degree of vertex v is a derivation of the degree of that vertex in graph G
       that does not take into account the vertices already belonging to set S(h-1) and the vertices adjacent to a vertex in set S(h-1).
       """
       return sum(1 for u in graph[v] if u in S and not any(w in S for w in graph[u]))
```

```
0s
     import random

     nodes=[5, 10, 15, 20, 25]
     prob=[0.25, 0.5, 0.75]

     def generate_random_graph(n, p):
       """
       Generates a random undirected graph with n vertices and an edge probability p
       Returns a dictionary with vertices as keys and adjacent vertices as values
       """
       graph = {i: set() for i in range(n)}  # Initialize graph with all vertices and no edges
       for i in range(n):
         for j in range(i+1, n):  # Only iterate over j > i to avoid adding duplicate edges
           if random.random() < p:
             graph[i].add(j)
             graph[j].add(i)  # Add edge in both directions
       return graph

     graphs=[]

     for node in nodes:
       for p in prob:
         graphs.append(generate_random_graph(node,p))

     for g in graphs:
       print(g)
```

```
# Printing the results
# The minimum dominating set is found by the Brute Force Algorithm
# The greedy dominating set is found by the Greedy Heuristic Algorithm
min_set1 = set()
min_set2 = set()
for i in range(15):
  min_set1 = greedy_algorithm(graphs[i])
  min_set2 = minimum_dominating_set(graphs[i])
  print("\nThe size of graph", i, "is: ", len(graphs[i]))
  print("\nMinimum Dominating Set: ", min_set2, "\tCardinality", len(min_set2))
  print("\nGreedy Dominating Set: ", min_set1, "\tCardinality", len(min_set1))
  print("----------")
  print()
```

5.4 The Results of the Initial Testing of the Heuristic Algorithm

The results of the initial testing of the heuristic algorithm using 15 sample inputs are summarized in Table 2 below. D denotes the minimum dominating set generated by the algorithm and |D| denotes its cardinality.

Table 2. Results of the Initial Testing of the Heuristic Algorithm

		Number of Vertices at Graph (n)				
		N=5	N=10	N=15	N=20	N=25
Edge Probability (p)	p=0.25	D={0,1,2} |D|= 3	D={0,1,2,3,4,5,8} |D|= 7	D={0,1,2,3,4,5,6,7,8,9,10,14} |D|= 12	D={0,1,2,3,4,5,6,11,13,14} |D|= 10	D={0,1,2,3,4,5,6,7,8,9,10,11,12,23} |D|= 14
	p=0.5	D={0,1,2,3} |D|= 4	D={0,1,2} |D|= 3	D={0,1,2,3,4,7} |D|= 6	D={0,1,2,19} |D|= 4	D={0,1,2,3,4,10} |D|= 6
	p=0.75	D={0,1} |D|= 2	D={0} |D|= 1	D={0,1} |D|= 2	D={0,4} |D|= 2	D={0,22} |D|= 2

6. EXPERIMENTAL ANALYSIS OF THE PERFORMANCE

As part of the performance testing, the performance of the implementation of the heuristic algorithm (described in Section 2.2) is analyzed experimentally. 50 runs are conducted for graphs with 50, 100, 200, 300, and 500 vertices. We kept the edge probability fixed as p=0.5 so that there is a single independent variable in the experiments, which is the input size of heuristic algorithm.

As a first step, to calculate the statistical measures such as mean running time, standard deviation, standard error, and 90% confidence interval, a function named

calculate_statistics(runtimes, num) is defined. As input, the function takes an array *runtimes*, which stores the runtime results of the 50 runs for graphs with 50, 100, 200, 300, and 500 vertices, in total having 250 entries, and an integer *num*, which denotes the number of sample runs for a certain input size. In this experiment, *num* is taken as 50 as we conducted 50 runs for each input size.

The experimental results from 50 runs, conducted for each input size with respectively 50, 100, 200, 300, and 500 vertices, are presented in Table 3. The table reports the mean time, standard deviation, standard error, and the 90% confidence level for each input size.

Table 3. Performance Testing Results from 50 Runs

Size	Mean Time(s)	Standard Deviation	Standard Error	90% CL
50	0.00124	0.00043	0.000061	(0.00114, 0.00134)
100	0.00566	0.00156	0.000221	(0.00529, 0.00603)
200	0.02380	0.00539	0.000763	(0.02253, 0.02506)
300	0.03907	0.01386	0.001960	(0.03582, 0.04232)
500	0.08102	0.01540	0.002177	(0.07741, 0.08464)

Table 3 demonstrates that as the input size increases, the mean running time of the heuristic algorithm increases. Furthermore, in Section 3.4, we showed that the theoretical time complexity of the algorithm is O(|E|*|V|^2) in the worst-case. If the graph is sparse, that is |E| = |V|, then the complexity becomes O(|E|*|V|^2) = O(|V|^3), however if the graph is dense, that is |E| = |V|^2, then the complexity becomes O(|E|*|V|^2) = O(|V|^4). The practical results of the greedy heuristic algorithm, as shown in Table 3, display that as the input size increases, mean time increases approximately in a linear fashion, which supports the theoretical result as the time complexity does not exceed O(|E|*|V|^2).

To better visualize the linear trend between the mean running time of the algorithm with the input size, the mean time results are plotted using a scatter chart in Excel. Furthermore, the best fit line for the measurement values is drawn on the chart to better understanding the nature of the linear relationship between the mean running time and the input size. The

best fit line is shown with the orange line. Figure 2 demonstrates the scatter plot with the best fit line for the measurement values:

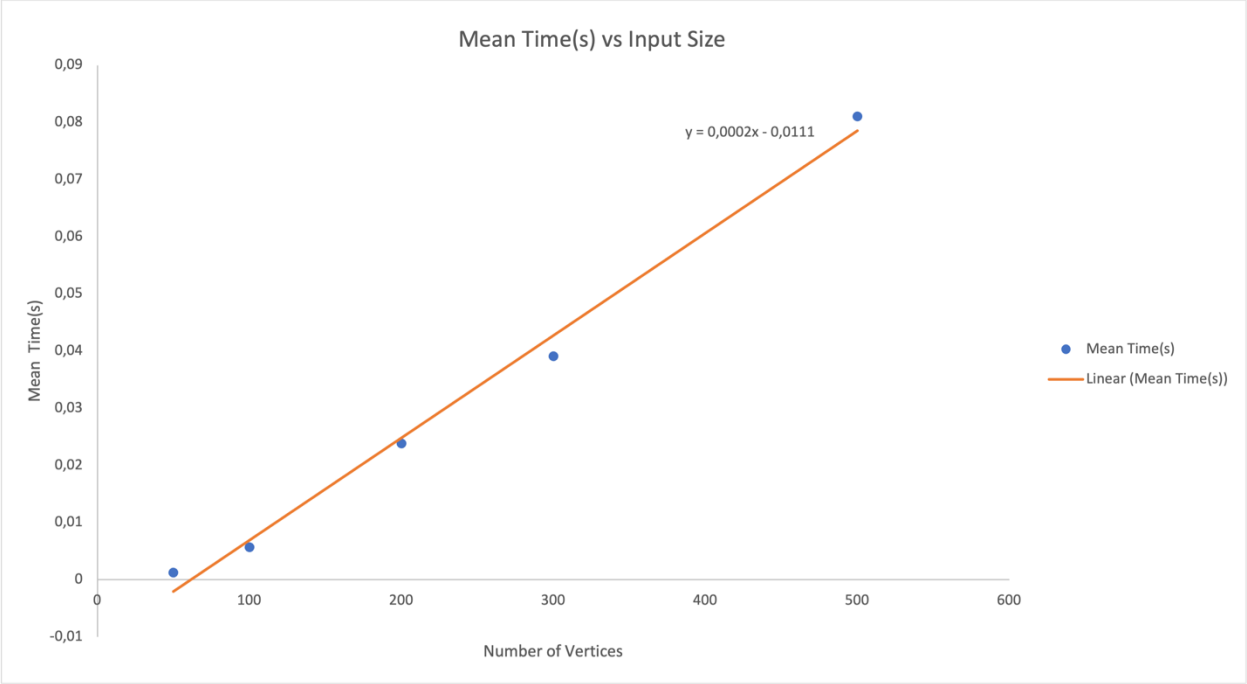


Figure 2. Scatter Plot of Mean Time(s) vs Input Size

7. EXPERIMENTAL ANALYSIS OF THE QUALITY

For the experimental analysis of the quality, we aim to find a relation between the exact solution and the solution that the heuristic algorithm provides to understand how close the solutions of the heuristic algorithm get to the exact answer. The relation will be defined with respect to the comparison of the cardinalities of the solutions generated by the brute force and heuristic algorithms. Like we did in the performance testing, as described in the previous section, we again kept the edge probability fixed as p=0.5 so that there is a single independent variable in the experiments, which is the input size of heuristic algorithm.

In the performed experiments, we first generated 20 graphs with varying input sizes. This began with a graph containing a single vertex, with the graph size incrementing by one vertex at a time until reaching a graph of 20 vertices. Then, the minimum dominating sets of these 20 graphs were determined using two methods: a brute force algorithm, which

produced the exact solution, and a heuristic algorithm, which generated an approximate solution. Finally, the cardinality results of these solutions were plotted in a bar chart to visualize the comparison between the exact and heuristic results, as shown in Figure 3:

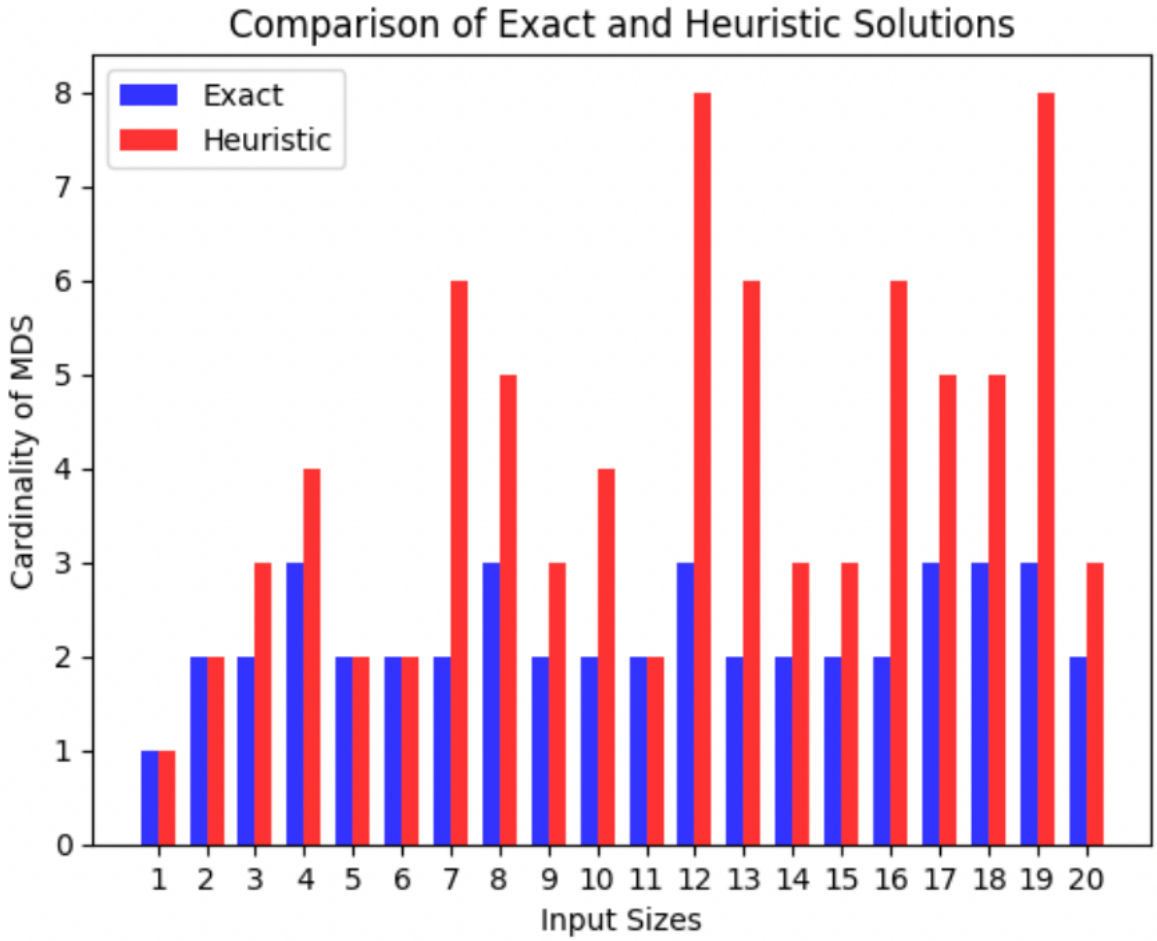


Figure 3. Comparison of Exact and Heuristic Solutions

In Figure 3, it is observed that there is a significant difference between the cardinalities of the exact and heuristic solutions. Thus, to further analyze how close the greedy heuristic algorithm get to the exact answer, we decided to draw another plot: a scatter plot showing the ratio of heuristic to exact solutions in terms of their cardinalities for each input size. The generated scatter plot is displayed in Figure 4:

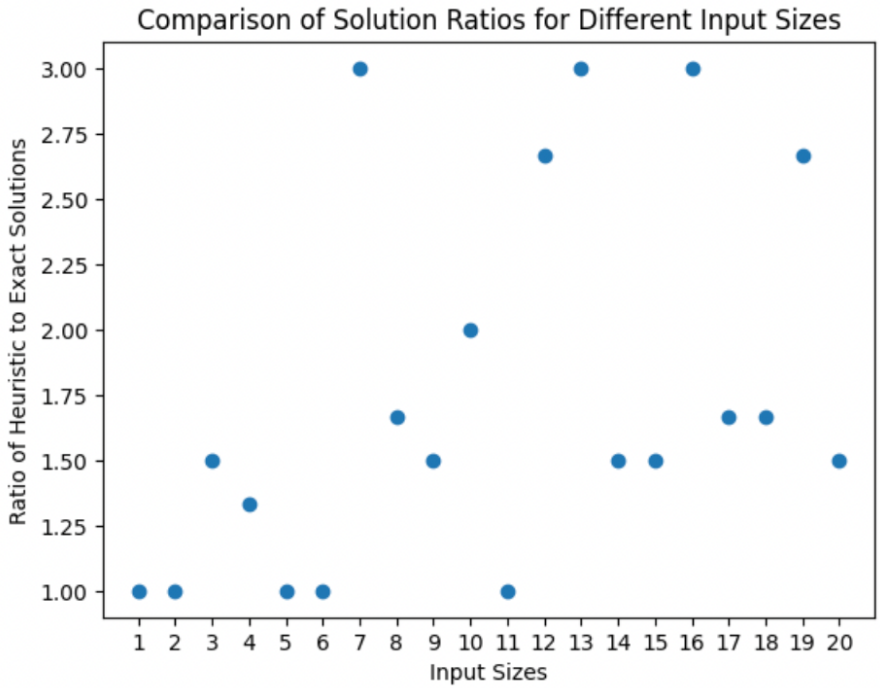


Figure 4. Comparison of Solution Ratios for Different Input Sizes

The scatter plot demonstrates the ratio bound $\rho(n) = 2$ does not hold for the experimental results since for input sizes 7, 12, 13, 16, and 19, the ratio of heuristic to exact solutions exceeds 2. Therefore, as also theoretically proven in Section 2.2, a constant ratio bound $\rho(n)$ cannot be derived for the greedy heuristic algorithm for the minimum dominating set problem.

8. EXPERIMENTAL ANALYSIS OF THE CORRECTNESS

For the experimental analysis of the correctness of the heuristic algorithm, we will use Black Box Testing. Since in the White Box Testing, line coverage is checked and since as long as the cardinality of the MDS is at least 2, White Box Testing always yields 100% statement coverage whereas in Black Box Testing, evaluation of algorithm in varying conditions is done, Black Box Testing yields a better experimental analysis of the correctness of the heuristic algorithm. Thus, as part of the Black Box Testing, we need to consider different scenarios in order to check the correctness of the heuristic algorithm under different conditions. The graph conditions upon which the algorithm will be checked can be found below:

- Graph with Single Vertex
- Complete Graph
- Disconnected Graph
- Random Graph with edge probability 50%

Graph with Single Vertex

Graphs consisting of one vertex must logically have only the corresponding vertex at their MDS, hence their cardinality must be equal to 1. To test this, we used the algorithm below, which checks whether logical correctness is parallel to the implementation-wise correctness.

```
lengths = []
num_vertices = []

for i in range(1, 10001):
    set1 = {i: set()}
    m=greedy_algorithm(set1)
    lengths.append(len(m))
    num_vertices.append(i)
```

When the graph comparing num_vertices and lengths were plotted, we reached to the plot below, which shows that the heuristic algorithm is correct for the graphs consisting of only 1 vertex since we acquired an MDS with cardinality 1 in all cases. The x-axis of the plot shows the graphs with single vertices, where the attributes of the vertices are different in each graph.

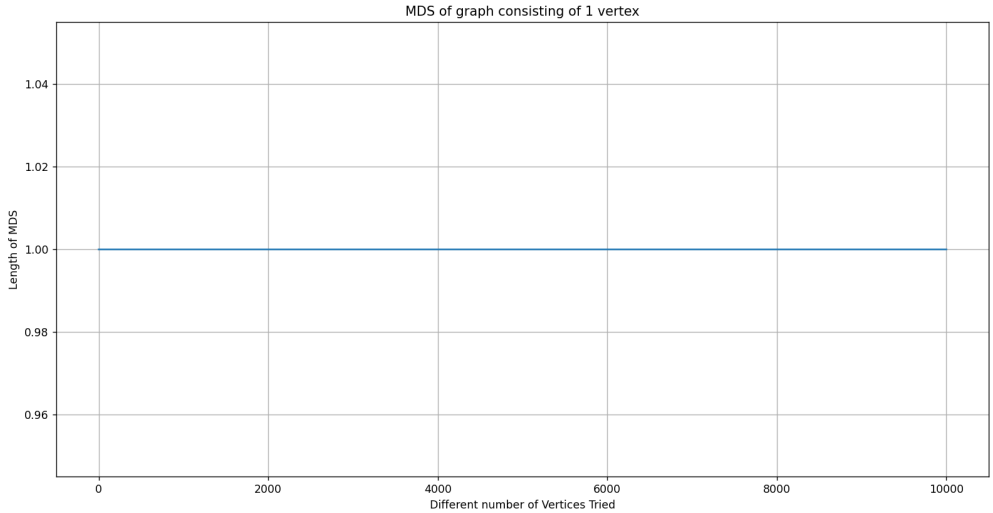


Figure 5. Minimum Dominating Sets of Graphs with Single Vertex

Complete Graph

Compelete graphs are graphs, where every vertex has |V|-1 edges and every vertex can be reached from any vertex via 1 edge only. Because of these properties, complete graphs must have an MDS with cardinality 1 since every vertex v ∈ V can be reached from any other vertex u ∈ V. To test this, we used the algorithm below, which checks whether logical correctness is parallel to the implementation-wise correctness in graphs with size ranging from 1 to 2000:

```
lengths = []
num_vertices = []
for n in range(1,2000):
    g = {v: set(range(n)) - {v} for v in range(n)}
    m=greedy_algorithm(g)
    lengths.append(len(m))
    num_vertices.append(n)
```

When the graph comparing num_vertices and lengths were plotted, we reached to the plot below, which shows that the heuristic algorithm is correct for the complete graphs since we acquired an MDS with cardinality 1 for all input sizes.

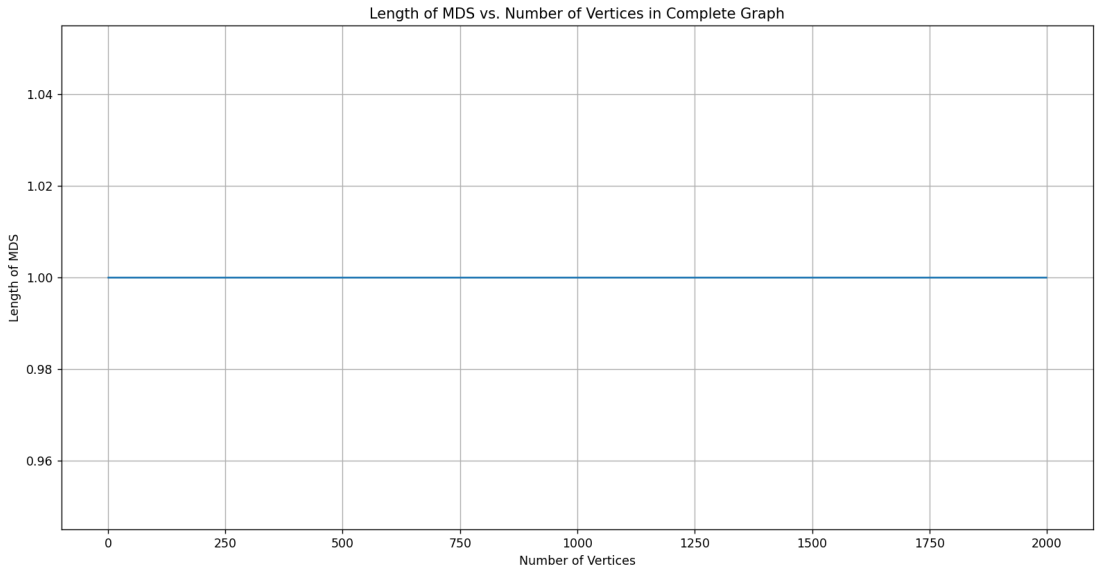


Figure 6. Minimum Dominating Sets of Complete Graphs

Disconnected Graph

An undirected graph is said to be disconnected if there exists vertices u, $v \in V$ such that there does not exist a path from u to v. Hence, cardinality of an MDS of a disconnected graph is at least 2, so vertices must be partitioned in a way such that there does not exist a path from vertex u to vertex v. In our case, we should check for the cases where cardinality of the MDS is equal to 2 and that happens when we partition both components having |V|/2 vertices. To test this argument, we used the algorithm below which checks whether logical correctness is parallel to the implementation-wise correctness with the input size ranging from 2 to 10000 (both inclusive):

```
lengths = []
num_vertices = []
for n in range(2,10001):
    for j in range(0, n, n//2):  # Number of vertices in each component
        g={}
        g[0] = set(range(1, j))
        g[j] = set(range(j+1, n+1))
        m=greedy_algorithm(g)
        if(len(m)==1): # when cardinality of m=1, there exists a node in the acyclic graph
            continue   # such that it has |V|-1 edges, hence it is not disconnected
        else:
            lengths.append(len(m))
            num_vertices.append(n)
```

This algorithm creates two MDS' for each number of vertices j. When MDS m= 1, it means that there exists a node in the acyclic graph such that it has |V|-1 edges, hence makes the graph connected. Therefore, we should analyze the MDS' which have cardinality >= 1. When we plot the graph comparing num_vertices and lengths, the plot below is reached, which shows that the heuristic algorithm is correct for the disconnected graphs with n/2 partitioning.

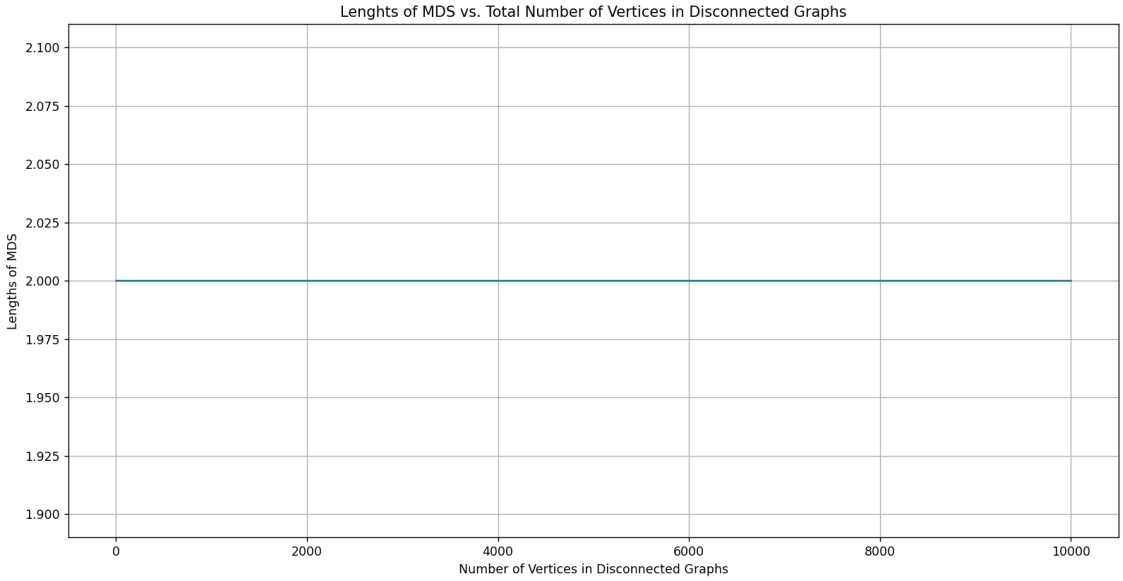


Figure 7. Minimum Dominating Sets of Disconnected Graphs

Random Graph with Edge Probability 50%

We generated the random graphs with 50% edge probability with our random graph generator algorithm. Our claim was that if the set S returned by our heuristic algorithm has cardinality <= 2, then the set S is the MDS of graph G. Hence, we created random graphs with number of vertices ranging from 1 to 20 (both inclusive). The distribution of the cardinalities of the set S for each graph can be found at the scatter plot below:

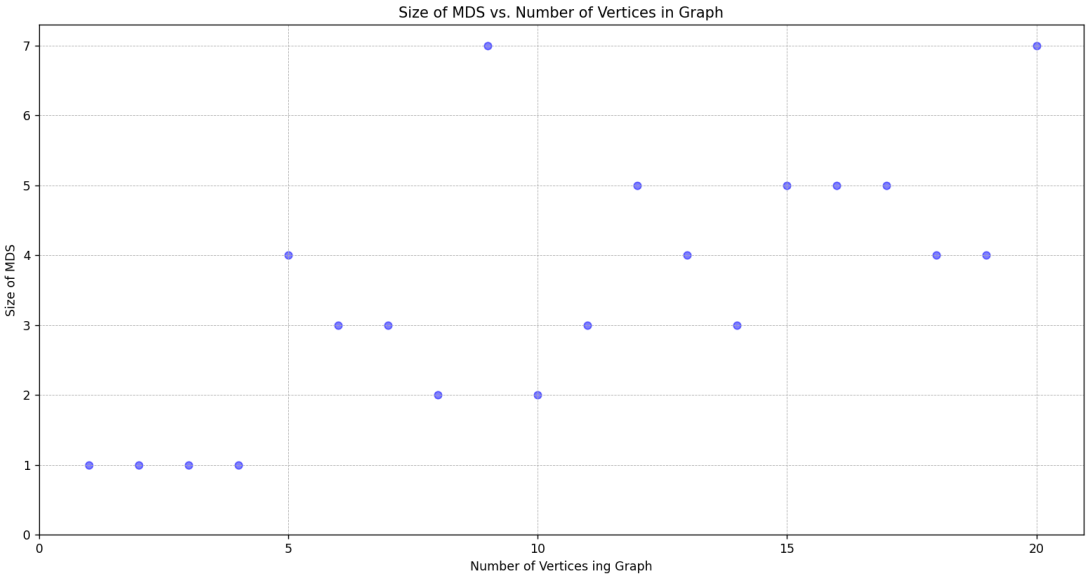


Figure 8. Size of MDS for Different Input Sizes

As it can clearly be seen from the plot, we need to analyze the graphs 1, 2, 3, 4, 8, and 10. Hence, we used the algorithm below to check whether set S was really the MDS of graph G or not.

The algorithm and its results can be found below:

```
lengths = []
num_vertices = []
graphs=[]

for n in range(1,21):
    graphs.append(generate_random_graph(n,0.5))

for i in range(len(graphs)):
    g=greedy_algorithm(graphs[i])
    m=minimum_dominating_set(graphs[i])
    lengths.append(len(g))
    num_vertices.append(len(graphs[i]))
    if(len(g)<=2 and len(g)==len(m)):
        print("\nGraph is : ", graphs[i])
        print("\n",g,"is a minimum dominating set with cardinality",len(g),"\n")
```

```
Graph is :  {0: set()}

 {0} is a minimum dominating set with cardinality 1


Graph is :  {0: {1}, 1: {0}}

 {0} is a minimum dominating set with cardinality 1


Graph is :  {0: {1, 2}, 1: {0}, 2: {0}}

 {0} is a minimum dominating set with cardinality 1


Graph is :  {0: {1, 2, 3}, 1: {0, 2, 3}, 2: {0, 1}, 3: {0, 1}}

 {0} is a minimum dominating set with cardinality 1


Graph is :  {0: {1, 2, 3, 5, 7}, 1: {0, 3, 4, 5, 6}, 2: {0, 3, 4, 5, 6, 7}, 3: {0, 1, 2, 5, 6}, 4: {1, 2, 6}, 5: {0, 1, 2, 3, 7}, 6: {1, 2, 3, 4, 7}, 7: {0, 2, 5, 6}}

 {0, 2} is a minimum dominating set with cardinality 2


Graph is :  {0: {8, 2, 6}, 1: {8, 9, 2, 3}, 2: {0, 1, 3, 4, 6, 7, 8}, 3: {1, 2, 5, 6, 7, 8, 9}, 4: {2, 5, 7, 8, 9}, 5: {8, 9, 3, 4}, 6: {0, 9, 2, 3}, 7: {8, 2, 3, 4}, 8: {0, 1, 2, 3, 4, 5, 7, 9}, 9: {1
, 3, 4, 5, 6, 8}}

 {8, 0} is a minimum dominating set with cardinality 2
```

The above result shows that the heuristic algorithm is correct for random graphs with 50% edge probability with MDS cardinality <=2.

In conclusion, we can state that the heuristic algorithm not only was correct theoretically, but also passed various tests for thousuands of graphs successfully in the following four categories: Graph with Single Vertex, Complete Graph, Disconnected Graph, and Random Graph with edge probability 50%.

9. DISCUSSION

In this report, we aimed to tackle an NP-hard problem, the Minimum Dominating Set problem, by formulating a brute-force algorithm and then, a heuristic algorithm. After theoretically analyzing both algorithms, we conducted experimental analysis for the heuristic algorithm to discuss its performance, solution quality, and correctness.

In Section 2.2, we theoretically showed that the greedy heuristic algorithm does not have a constant ratio bound. To further prove the lack of a ratio bound experimentally, we compared the exact and heuristic results in Section 7, as part of the experimental analysis of the quality. While testing our greedy heuristic algorithm on graphs with various characteristics, we came to the conclusion that the heuristic algorithm does not have a strict ratio bound in practice, which can be visually seen in Figures 3 and 4.

In Section 3.3, as part of the correctness analysis of the heuristic algorithm, it was shown that the greedy heuristic algorithm was theoretically optimal if the set it returns has a cardinality less than or equal to 2. Then, in Section 8, we tested whether the greedy heuristic algorithm works optimally in practice and it was seen that the algorithm works correctly under 4 different graph categories which are Graph with Single Vertex, Complete Graph, Disconnected Graph, and Random Graph with edge probability 50%.

Furthermore, the experimental analysis of the performance in Section 6 demonstrated that as the input size increases, the mean running time of the heuristic algorithm increases approximately linearly, which was a trend we presumed.

In conclusion, we did not observe any inconsistencies between the theoretical and experimental results we acquired. Hence, it can be claimed that our greedy heuristic algorithm works correctly in solving the minimum dominating set problem, without showing any obvious defects.

10. REFERENCES

Casado, A., Bermudo, S., López-Sánchez, A. D., & Sánchez-Oro, J. (2023). An iterated greedy algorithm for finding the minimum dominating set in graphs. Mathematics and Computers in Simulation, 207, 41-58. https://doi.org/10.1016/j.matcom.2022.12.018.

Chlebík, M., & Chlebíková, J. (2008). Approximation hardness of dominating set problems in bounded degree graphs. Information & Computation, 206(11), 1264–1275.

Feige, U. (1998). On the hardness of approximating minimum vertex cover. Journal of the ACM, 45(2), 433-438.

Figure 1. Wikipedia. (2023). Dominating set [Image]. Retrieved from https://en.wikipedia.org/wiki/Dominating_set [Accessed April 26, 2023].

Hernández Mira, F. Á., et al. (2022). A Polynomial-Time Approximation to a Minimum Dominating Set in a Graph. Theoretical Computer Science, 930, 142-156. https://doi.org/10.1016/j.tcs.2022.07.020.

Hochbaum, D. S., Rossmanith, P., & Zangl, K. (2005). Approximation algorithms for NP-hard problems (pp. 201-241). Springer.