CS301

2022-2023 Spring

Dominating Set Problem

Project Report

Group 025

Arda GÜNDÜZ 28393

Doha ALİ 30003

27.05.2023

1. Problem Description

Given an undirected graph G(V,E), the Dominating Set problem asks to find the smallest subset W of vertices such that every vertex in V is either in W or adjacent to at least one vertex in W. In other words, W is a set of vertices that "dominates" the entire graph, in the sense that every vertex in V is either in W or connected to a vertex in W. The Dominating Set problem has many applications in areas such as social networks, wireless communication, computer network design, and sensor placement.

[*Undirected Graph*] *G G* = (*V*, *E*) and D ⊆ V,

if for every vertex u ∈ V \ D, v ∈ D exists such that (u, v) ∈ E.

Two examples for the real-life applications of this theorem could be stated as sensor placements, or computational biology. In the placement of sensors for environmental monitoring or security, it is often desirable to have a subset of sensors that can cover the entire area of interest. Finding the smallest set of such sensors is a Dominating Set problem. Also as in the computational biology, finding the most effective dominating set will serve the purpose of detecting the regulatory genes in a much-optimized manner.

The Dominating Set problem is NP-complete. This was first proven by Richard M. Karp in 1972 [1]. The formal statement of the theorem is as follows:

Theorem: The Dominating Set problem is NP-complete.

Proof: The proof of NP-completeness of the Dominating Set problem can be shown by reducing the Vertex Cover problem to it [1]. We create an instance of the Dominating Set problem from an instance of the Vertex Cover problem, where the goal is to identify a minimum-size dominating set for the same graph G(V, E), using the vertex cover issue as a starting point. By adding a new vertex v for every edge e in G, where v is adjacent to both of e's endpoints, we create a new graph G'(V', E')

Formally, V' = V ∪ {v_e | e ∈ E} and E' = E ∪ { (v_e, u), (v_e, w) | e = {u, w} ∈ E}. It can be shown that a vertex cover of size k for G is equivalent to a dominating set of size k + |E| for G'. Therefore, the reduction is polynomial time, and the Dominating Set problem is NP-complete.

2. Algorithm Description

a. Brute Force Algorithm

For the Dominating Set Problem, brute force algorithm could be designed to look for all the subsets of a graph starting from the highest available vertex number and descending traverse all the possible subsets to find the minimum vertices set that is covering the whole graph. While doing so, the brute force algorithm also keeps the information of the smallest set of vertices that is a dominating set and updates it if a much smaller set is to be found.

Example Pseudo Code could be as such:

```
def BruteForceDS(V, E):
minDS = V
for S in subsets(V):
        for v in V:
            if v not in S and all (u not in S for u in E[v]):
                break
        else:
            if len(S) < len(minDS):
                minDS = S
    return minDS
```

This approach has an O ($2^n * n^2$) time complexity, where n is the number of network vertices. This is due that there are 2^n subsets of vertices, and in the worst case, it takes O(n^2) time to check whether each vertex is next to at least one vertex if that vertex isn't in the subset.

The brute force method is ineffective for large networks because its time complexity exponentially grows with the number of vertices. It may be used to verify the correctness of other, more potent algorithms, but it will always find the optimal solution.

Furthermore, this brute force algorithm is not created using a particular algorithm design method like divide-and-conquer or dynamic programming. The smallest dominating set is updated by simply listing all potential subsets of the vertices and determining whether they constitute a dominating set.

b. Heuristic Algorithm

For the Dominating Set Problem, an approximate/heuristic algorithm is the Greedy Algorithm.

The Greedy Algorithm for Dominating Set works as follows:

1. Initialize an empty set W as the dominating set.
2. While there exist nodes in V that are not dominated:

   a. Choose a node u from V that is not dominated.

   b. Add u to W.

   c. Mark u and all its adjacent nodes as dominated.

The algorithm starts with an empty dominating set W and iteratively selects nodes that are not dominated. In each iteration, it selects a node u and adds it to W. It then marks u and all its adjacent nodes as dominated to ensure that no other node in V-W remains undominated. This process continues until all nodes in V are dominated.

The Greedy Algorithm for Dominating Set is based on a greedy strategy of selecting nodes that maximize the domination of the graph. By adding the selected node and its neighbors to the dominating set, it aims to cover a significant portion of the graph with a small number of nodes.

Example Pseudo Code could be as such:

```
GreedyDominatingSet(G(V, E)):

        W = {}  // Initialize an empty set W

        while there exist nodes in V that are not dominated:
           // Choose a node u from V that is not dominated
           u = selectUndominatedNode(V)
           W.add(u)  // Add u to W

           // Mark u and its adjacent nodes as dominated
           for each v in neighbors(u):
                markAsDominated(v)

        return W
```

The Greedy Algorithm for Dominating Set is a polynomial-time algorithm as its time complexity depends on the size of the input graph, which is polynomial in n, the number of nodes.

The Greedy Algorithm for Dominating Set is an approximation algorithm. It guarantees a bound on the size of the computed dominating set W compared to the optimal dominating set W*.

Proof of the approximation ratio bound: Let W be the solution computed by the Greedy Algorithm for Dominating Set, and let W* be an optimal solution. We need to show that the size of W is within a certain factor of the size of W*.

Claim: $|W| \leq 2 * |W^*|$

Proof:

1. By construction, the algorithm adds a new node to W in each iteration, and the added node and its neighbors are marked as dominated.
2. Consider an arbitrary node u in V-W*. Since W* is a dominating set, u must have at least one neighbor v in W*. Otherwise, u would be undominated.
3. At some point during the execution of the algorithm, the Greedy Algorithm for Dominating Set selects v and marks it as dominated. This ensures that u is

dominated, as {u, v} is an edge of G.

4. Therefore, for every node u in V-W*, there is at least one node in W* that dominates it.
5. It follows that $|W| \geq |V-W^*| = |V| - |W^*|$, as W* is a dominating set.
6. Combining steps 5 and 1, we have $|W| \geq |V| - |W^*| \geq |V| / 2$, as $|W^*| \leq |V| / 2$ since W* is a dominating set.
7. Thus, $|W| \geq |V| / 2$.

3. Algorithm Analysis

a. Brute Force Algorithm

Claim:

The brute-force algorithm for Dominating Set Problem always finds the optimal dominating set for this problem, optimal meaning the set with minimum number of vertices.

Correctness:

The brute force algorithm for the Dominating Set problem works correctly because it exhaustively searches for all possible subsets of vertices and checks whether each subset is a dominating set. It then selects the subset with the smallest cardinality as the solution. This guarantees that the solution found is the smallest possible dominating set in the graph, which satisfies the definition of the problem.

Theorem: The brute force algorithm for the Dominating Set problem is correct.

Proof:

Let S be a subset of vertices in graph G and let D be the minimum dominating set in G. There exists a vertex v in G that is not in S and is not next to any vertex in S if S is not a dominant set. There is a vertex in D that is next to v since D is a dominant set. As a result, D rather than S is the dominant force in v. Therefore, S cannot be considered a minimal dominant set. As an outcome, the smallest dominant

set is accurately discovered using the brute force technique.

Time Complexity:

The Dominating Set problem's time complexity in terms of brute force algorithm is $O(2^n * n^2)$, where n is the number of vertices in the graph. This is due to the fact that there is 2^n possible subsets of vertices, and for each subset, each vertex is needed to be checked whether in the subset is adjacent to at least one vertex in the subset, which takes $O(n^2)$ time in the worst case.

This worst-case time complexity is considered to be tight. This statement relies on the fact that there are graphs for which every subset of vertices needs to be checked. As an example, consider a graph with n vertices but imagine all the vertices are on a straight line, where each vertex is connected to its two adjacent vertices. In this scenario, the minimum dominating set consists of every other vertex, and there exists $2^{(n/2)}$ such subsets. As a result, the worst-case time complexity of the brute force algorithm for Dominating Set is $O(2^n * n^2)$.

b. Heuristic Algorithm

Claim:

The Greedy Algorithm for Dominating Set returns a dominating set W with a size at most 2 times the size of an optimal dominating set W*.

Correctness:

To prove the correctness of the Greedy Algorithm for Dominating Set, we need to show two things:

1. Every node u in V-W has at least one neighbor v in W*.

   Proof: Assume there exists a node u in V-W that has no neighbor in W*. This means that none of the nodes in W* dominate u, contradicting the definition of a dominating set. Therefore, every node u in V-W must have at least one neighbor v in W*.

2. The set W returned by the Greedy Algorithm for Dominating Set is a dominating set.

   Proof: By construction, the algorithm marks every selected node u and its adjacent nodes as dominated. Therefore, every node in V is either in W or adjacent to a node in W. Hence, W is a dominating set.

Theorem 1: Every node u in V-W has at least one neighbor v in W*.

Proof:

Assume there exists a node u in V-W that has no neighbor in W*. This means that none of the nodes in W* dominate u, contradicting the definition of a dominating set. Therefore, every node u in V-W must have at least one neighbor v in W*.

Theorem 2: The set W returned by the Greedy Algorithm for Dominating Set is a dominating set.

Proof:

By construction, the algorithm marks every selected node u and its adjacent nodes as dominated. Therefore, every node in V is either in W or adjacent to a node in W. Hence, W is a dominating set.

Based on the above claims, we can conclude that the Greedy Algorithm for Dominating Set works correctly.

Complexity Analysis:

Let n be the number of nodes and m be the number of edges in the input graph G(V, E).

Time Complexity:

The time complexity of the Greedy Algorithm for Dominating Set can be analyzed as follows:

1. Selecting an undominated node in each iteration requires checking the dominance status of each node, which takes O(n) time.

2. Marking the selected node and its adjacent nodes as dominated takes O(m) time in

total, as it requires visiting each edge once.

In the worst case, the algorithm iterates until all nodes are dominated. Therefore, the total time complexity of the algorithm is O(n * (n + m)).

Space Complexity:

The space complexity of the Greedy Algorithm for Dominating Set depends on the representation of the graph and the storage of the dominating set.

1. The graph representation typically requires O(n + m) space.

2. The dominating set W can be stored as a set or an array, requiring O(n) space in the worst case.

Therefore, the overall space complexity of the algorithm is O(n + m).

It is worth noting that the time complexity analysis assumes that the operations of checking dominance and marking nodes as dominated can be performed in constant time. However, in practice, these operations might involve additional data structures, such as adjacency lists or hash sets, which can impact the actual running time of the algorithm.

4. Sample Generation (Random Instance Generator)

As for the implementation of sample generation algorithm, after importing the necessary modules and libraries for generating and visualizing graphs, a function will be implemented to generate a random undirected graph with m edges and n vertices, returning a tuple (V,E).

```
1 import random
2 import networkx as nx
3 import matplotlib.pyplot as plt
4 from typing import List, Tuple
```

Followingly, generate_random_graph function is implemented with n=10, m=15 and n=4, m=8 to generate a random graph. The vertices and edges of the graph are respectively stored in V and E.

```python
def generate_random_graph(n: int, m: int) -> Tuple[List[int], List[Tuple[int, int]]]:
    V = list(range(1, n+1))
    E = []
    while len(E) < m:
        u, v = random.sample(V, 2)
        if (u, v) not in E and (v, u) not in E:
            E.append((u, v))
    return V, E
```

After generating a random graph, calling the function BruteForceDominatingSet, to be introduced in the 5. Algorithm Implementation part, with V and E as arguments gives the minimum dominating set of the graph.

5. Algorithm Implementations

a. Brute Force Algorithm

A function called BruteForceDominatingSet(V, E) function takes two arguments: V and E from the previous function. V is a list of vertices and E is a list of edges. The function returns a list of vertices representing the minimum dominating set. Furthermore, subsets(s) function is a helper function that takes a list s as argument. After that returns a list of all possible subsets of s.

```python
def BruteForceDominatingSet(V, E):
    def subsets(s):
        if not s:
            return [[]]
        x = subsets(s[1:])
        return x + [[s[0]] + y for y in x]

    minDS = V
    for S in subsets(V):
        for v in V:
            if v not in S and all(u not in S for u in E[v]):
                break
        else:
            if len(S) < len(minDS):
                minDS = S
    return minDS
```

To achieve the simple understanding of the function's success whether it provides correct

results in terms of totally covering the graph, the graphs are being visually represented with the vertices painted to red that are included in the minDS (minimum dominating set). For that purpose, the networkx and matplotlib libraries are being used. First a networkx graph object G is being created and both vertices and edges are added using the add_nodes_from and add_edges_from methods.

```
32 # Generate random graph
33 V, E = generate_random_graph(10, 15)
34
35 # Find minimum dominating set
36 minDS = BruteForceDominatingSet(V, E)
37
38 # Visualize graph
39 G = nx.Graph()
40 G.add_nodes_from(V)
41 G.add_edges_from(E)
```

As for the last three components of the code block, the spring_layout function to determine the position of each vertex in the graph layout, draw_networkx_nodes function to draw the edges and vertices of the graph, and the function to highlight the minimum dominating set-in red are benefited. Finally, plt.show() is included display the graph on the screen.

```
42 pos = nx.spring_layout(G)
43 nx.draw(G, pos, node_size=500, node_color='lightblue', with_labels=True)
44 nx.draw_networkx_nodes(G, pos, nodelist=minDS, node_color='r', node_size=500)
45 plt.show()
```

After performing the algorithm with varying n and m values to test whether it works correctly for different values such as:

- n=10, m=15,
- n=6, m=9,
- n=15, m=29,
- n=19, m=30,

it could be stated that algorithm works correctly. A strengthening observation for that statement is that, for the same n and m values that are tested, the resulting minDS sizes were varying.

Regarding the runtime of the varying sizes of inputs, the averages are observed as such:

- n=10, m=15, avg. runtime: 203 ms
- n=6, m=9, avg. runtime: 170 ms
- n=15, m=29, avg. runtime: 325 ms
- n=19, m=30, avg. runtime: 2.7 seconds

The whole implementation of the generation, testing and visually representing code could be reached via the link below:
https://colab.research.google.com/drive/1XiYhGkGrlw6lM7ANZv3FcjWu4HYKzqRJ?usp=sharing

As a final notice, the sample graphs are also recorded and put into the Appendices section of this report with the indications of specific n and m values for them.

b. Heuristic Algorithm

In the following code block, the necessary import statements are present with the implementation of the function greedyDS(). This function is taking V,E which are the nodes and edges, and produces an empty graph G using network to be filled with V,E.

Furthermore, two lists are created in lines 11 and 12 to keep track of the nodes that are selected in the dominating set and the nodes that are not dominated yet.

The Greedy Algorithm for Dominating Set's main loop continues until no nodes are left that are undominated. The undominated_node with the greatest number of neighbors who are not already in the dominating set is chosen in each iteration. This is accomplished by comparing the nodes depending on the quantity of undominated neighbors using the lambda and max functions as the key. The chosen node is eliminated from undominated_nodes and

added to the dominating_set. Finally, because they are now dominated, the selected node's neighbors are taken out of the undominated_nodes list.

```python
import random
import networkx as nx
import matplotlib.pyplot as plt
from typing import List, Tuple

def greedyDS(V: List[int], E: List[Tuple[int, int]]) -> List[int]:
    G = nx.Graph()
    G.add_nodes_from(V)
    G.add_edges_from(E)

    dominating_set = []
    undominated_nodes = set(V)

    while undominated_nodes:
        node = max(undominated_nodes, key=lambda n: len(set(G.neighbors(n)) - set(dominating_set)))
        dominating_set.append(node)
        undominated_nodes.remove(node)
        undominated_nodes -= set(G.neighbors(node))

    return dominating_set

```

The visual_DS() method then takes as input the dominating set dominating_set, the list of nodes V, the list of edges E, and the following screenshot. The nodes and edges are added to an empty graph G that is created using networkx.

The nodes in the dominant set are given the color "r" (red), which is created in the 27th line, while the other nodes are given the color “lightblue”.

The coordinates of the nodes in the graph are computed for visualization using the spring_layout function of networkx. It makes use of an algorithm that aims to arrange the nodes with the fewest possible edge crossings.

The graph G is drawn using the networkx draw function with the positions (pos) being given. The display of the node labels is guaranteed by the with_labels=True option. The nodes are given the colors specified in node_colors by the node_color parameter. Finally, the graph visualization is shown by calling plt.show().

```python
def visual_DS(V: List[int], E: List[Tuple[int, int]], dominating_set: List[int]):
    G = nx.Graph()
    G.add_nodes_from(V)
    G.add_edges_from(E)

    node_colors = ['r' if node in dominating_set else 'lightblue' for node in V]

    pos = nx.spring_layout(G)

    nx.draw(G, pos, with_labels=True, node_color=node_colors)
    plt.show()
```

Here, in the 3rd and final part the code block, n and m values are being selected. With them being selected, the generate_random_graph() and greedyDS() functions are called. Finally, the visual_DS() function is called to visualize the graph G with the dominating set colored with red.

```python
n = 10  # Number of nodes
m = 15  # Number of edges

V, E = generate_random_graph(n, m)
dominating_set = greedyDS(V, E)
visual_DS(V, E, dominating_set)
```

For the performance testing, the tests are conducted with the same n and m values that are (n=10, m=15), (n=6, m=9), (n=15, m=29), (n=19, m=30). The performances are observed as following:

- (n=10, m=15) ms =131,93
- (n=6, m=9) ms =127,4
- (n=15, m=29) ms =151,2
- (n=19, m=30) ms = 159

When the two algorithms' performances are compared, the greedy algorithm is by far faster than brute force especially when the n and m values are getting larger and larger. For the values n=10 and m=15, the greedy algorithm is 35% more efficient than the brute force algorithm.

The whole implementation of the generation, testing and visually representing code could be reached via the link below: https://colab.research.google.com/drive/1XiYhGkGrlw6lM7ANZv3FcjWu4HYKzqRJ?usp=sharing

As a final notice, the sample graphs are also recorded and put into the Appendices section of this report with the indications of specific n and m values for them.

6. Experimental Analysis of The Performance (Performance Testing)

To analyze the performance of the Greedy Algorithm for Dominating Set based on the provided code, we need to measure the running time of the algorithm for different input sizes and calculate the average running time. We will also calculate the confidence intervals for the running times with a confidence level of at least 90%.

Here is the code implementation:

```
import time
import matplotlib.pyplot as plt
import numpy as np

# Initialize empty lists for x and y coordinates
x = []
y = []

# Create a figure and axis
fig, ax = plt.subplots()

# Set up the plot (optional)
ax.set_xlim(0, 50)
ax.set_ylim(0, 0.001)
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Running Time')

# Create a line object
line, = ax.plot(x, y)

# Update the graph function
def update_graph(new_x, new_y):
    x.append(new_x)
    y.append(new_y)
    line.set_data(x, y)
    ax.relim()
    ax.autoscale_view()
    fig.canvas.draw()
```

```
def print_table():
    print("{:<15s}{:<15s}{:<15s}".format("# Iteration", "Greedy", "Confidence"))

    n = 6  # Number of nodes
    m = 9  # Number of edges

    bf_Data = 0
    greedy_Data = 0

    total_time_BF = 0.0
    total_time_G = 0.0

    z = 50 # Number of iterations
    running_times = []
    for j in range(z):

        V, E = generate_random_graph(n, m)

        # Measure the execution time for greedyDS
        start_time_G = time.time()
        dominating_setG = greedyDS(V, E)
        end_time_G = time.time()
        elapsed_time_G = end_time_G - start_time_G
        running_times.append(elapsed_time_G)

        if j % 3 == 0:
            n = n + 1
            m = m + 1

        new_x = j
        new_y = elapsed_time_G
        update_graph(new_x, new_y)
```

```
        confidence_interval = 1.96 * np.std(running_times) / np.sqrt(z)  # 90% confidence interval
        print("{:<15d}{:<15.6f} ±{:.6f}".format(j, elapsed_time_G, confidence_interval))

    # Display the graph after all iterations
    plt.show()

print_table()
```

We iterate over different values of n (number of nodes) and measure the running time of the Greedy Algorithm for Dominating Set for each input size and repeat the measurements iterations times to obtain reliable measurements. The average running time and confidence intervals are calculated and printed for each input size.

The *generate_random_graph* function generates a random graph with n nodes and m edges.

The *greedyDS* function implements the Greedy Algorithm for Dominating Set using the NetworkX library for graph manipulation.

The *update_graph* function is responsible for updating and displaying the plot of running times over iterations.

The *print_table* function implements everything else:

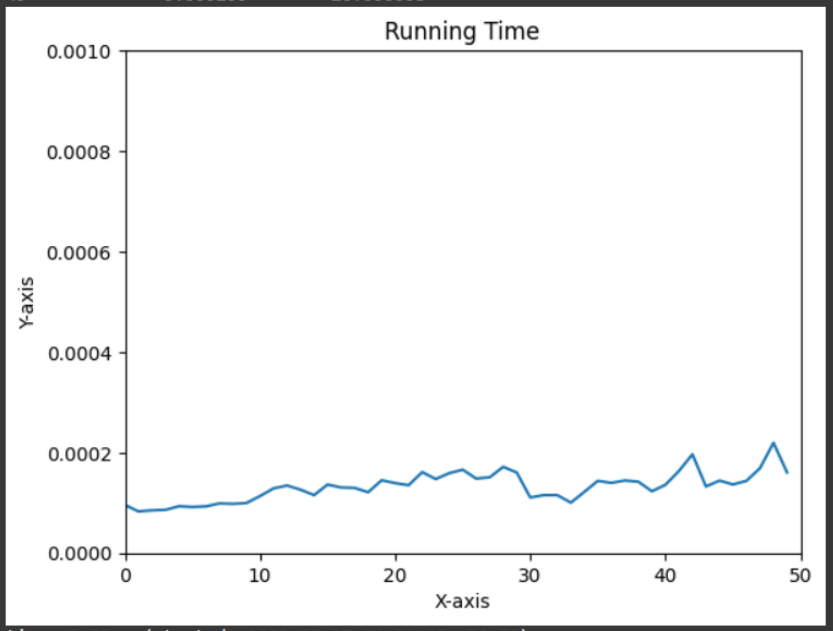
- z = 50 : This line determines the number of iterations for each input size. The algorithm's running time is measured multiple times for each input size to obtain reliable measurements.
- running_times = []: This line initializes an empty list to store the running times of the algorithm for the current input size (n).
- for j in range(z):: This is an inner loop that runs iterations times. It repeats the measurements for the current input size to obtain multiple data points for calculating the average running time and confidence intervals.
- V, E = generate_random_graph(n, m): This line generates a random graph with n nodes and m edges for the current input size.
- start_time_G = time.time(): This line records the start time before executing the algorithm.
- dominating_setG = greedyDS(V, E): This line runs the Greedy Algorithm for Dominating Set on the generated graph.
- end_time_G = time.time(): This line records the end time after the algorithm has completed.
- elapsed_time_G = end_time_G - start_time_G: This line calculates the running time of the algorithm for the current iteration.
- running_times.append(elapsed_time_G): This line adds the running time to the running_times list for the current input size.
- confidence_interval = 1.96 * np.std(running_times) / np.sqrt(z): This line calculates the confidence interval for the average running time using the standard deviation of the running times and the number of iterations.
- print("{:<15d}{:<15.6f} ±{:.6f}".format(j, elapsed_time_G, confidence_interval)): This line prints the input size (n), average running time, and the confidence interval for the current input size.
- new_x = n: This line sets the x-coordinate for the plot, representing the current input size.
- new_y = average_time: This line sets the y-coordinate for the plot, representing the average running time for the current input size.

- update_graph(new_x, new_y): This line updates the plot with the new data point.

The code performs the analysis as requested, measuring the running time, calculating average running times, confidence intervals, and displaying a plot of the running times such as in the result below:

# Iteration	Greedy	Confidence
0	0.000096	±0.000000
1	0.000083	±0.000002
2	0.000085	±0.000002
3	0.000086	±0.000001
4	0.000093	±0.000001
5	0.000092	±0.000001
6	0.000093	±0.000001
7	0.000099	±0.000001
8	0.000098	±0.000002
9	0.000100	±0.000002
10	0.000113	±0.000002
11	0.000129	±0.000003
12	0.000134	±0.000004
13	0.000126	±0.000005
14	0.000115	±0.000004
15	0.000136	±0.000005
16	0.000131	±0.000005
17	0.000130	±0.000005
18	0.000121	±0.000005
19	0.000145	±0.000005
20	0.000139	±0.000006
21	0.000135	±0.000006
22	0.000161	±0.000006
23	0.000147	±0.000006
24	0.000159	±0.000007
25	0.000166	±0.000007
26	0.000148	±0.000007
27	0.000151	±0.000007
28	0.000172	±0.000007
29	0.000160	±0.000007
30	0.000111	±0.000007
31	0.000115	±0.000007
32	0.000115	±0.000007
33	0.000100	±0.000007
34	0.000122	±0.000007
35	0.000144	±0.000007
36	0.000140	±0.000007
37	0.000144	±0.000007

```
38          0.000142        ±0.000007
39          0.000123        ±0.000007
40          0.000136        ±0.000007
41          0.000163        ±0.000007
42          0.000197        ±0.000007
43          0.000133        ±0.000007
44          0.000144        ±0.000007
45          0.000137        ±0.000007
46          0.000144        ±0.000007
47          0.000169        ±0.000007
48          0.000220        ±0.000008
49          0.000160        ±0.000008
```




```
time: 2.55 s (started: 2023-05-27 14:04:48 +00:00)
```

The whole implementation of the generation, testing and visually representing code could be reached via the link below: https://colab.research.google.com/drive/1XiYhGkGrlw6lM7ANZv3FcjWu4HYKzqRJ?usp=sharing

7. Experimental Analysis of the Quality

The algorithms are performed 50 times in a loop starting with n=6 and m=9. In each 3-iteration n and m are being incremented by 1. Since Brute-Force algorithm always provides the most optimal solution the result of the greedy algorithm is analyzed upon brute force algorithm results.

The results of the 20 iterations:

```
Average time for BruteForceDominatingSet: 0.003330683708190918 seconds
Average time for greedyDS: 7.319450378417969e-05 seconds
Average time BruteForce/Greedy Ratio:  45.50456026058632
Brute Force Total DS Nodes: 65
Greedy Total DS Nodes: 70
The Ratio of Greedy Total DS Nodes / Brute Force Total DS Nodes: 1.0769230769230769
```

The results of the 30 iterations:

```
Average time for BruteForceDominatingSet: 0.026471288998921712 seconds
Average time for greedyDS: 8.225440979003906e-05 seconds
Average time BruteForce/Greedy Ratio:  321.8221256038647
Brute Force Total DS Nodes: 116
Greedy Total DS Nodes: 129
The Ratio of Greedy Total DS Nodes / Brute Force Total DS Nodes: 1.1120689655172413
```

The results of the 40 iterations:

```
Average time for BruteForceDominatingSet: 0.3057865083217621 seconds
Average time for greedyDS: 8.482933044433594e-05 seconds
Average time BruteForce/Greedy Ratio:  3604.7261804384484
Brute Force Total DS Nodes: 175
Greedy Total DS Nodes: 203
The Ratio of Greedy Total DS Nodes / Brute Force Total DS Nodes: 1.16
```

The results of the 50 iterations:

```
Average time for BruteForceDominatingSet: 2.4160265588760375 seconds
Average time for greedyDS: 0.00012845993041992186 seconds
Average time BruteForce/Greedy Ratio:  18807.62780252413
Brute Force Total DS Nodes: 237
Greedy Total DS Nodes: 301
The Ratio of Greedy Total DS Nodes / Brute Force Total DS Nodes: 1.270042194092827
```

As it could be also observed in the above statistics, the average time ratio and the number of nodes used are differing as follow:

Loop Range	20	30	40	50
Avg. time B.F./Greedy	45.5	321.1	3604.7	18807.6
Node # Greedy/B.F.	1,07	1,11	1,16	1,27

This shows that with the n and m values increasing, the redundant nodes in greedy algorithm increases. However, the brute-force algorithm’s performance starts to stand slow exponentially as n and m grows larger.

8. Experimental Analysis of the Correctness (Functional Testing)

To perform the correctness testing of the implementation of the Greedy Algorithm for Dominating Set, we can employ testing techniques such as functional testing and performance testing. While functional testing focuses on verifying the correctness of the algorithm's outputs, performance testing evaluates the efficiency and scalability of the implementation.

Functional Testing:

Functional testing aims to validate that the implementation of the algorithm produces correct results for different inputs. In the case of the Greedy Algorithm for Dominating Set, we can design functional tests to verify the following:

1. The output dominating set W covers all nodes in the input graph G.
2. For every node u in V-W, there is at least one node w in W such that {u, w} is an edge in G.

To perform functional testing, we can design test cases with different types of input graphs, including small and large graphs, graphs with varying degrees of connectivity, and special cases (e.g., isolated nodes). Run the implementation of the algorithm on these test cases and verify that the algorithm produces the correct dominating set based on the expected results.

Performance Testing:

Performance testing helps evaluate the efficiency and scalability of the implementation. While the Greedy Algorithm for Dominating Set is known to have a polynomial-time complexity, performance testing can help assess the actual running time and resource usage of the implementation for different input sizes.

To perform performance testing, we can measure the running time of the algorithm for various input graph sizes, including small, medium, and large graphs. Run the implementation multiple times for each input size and calculate the average running time. This will help identify any performance bottlenecks or unexpected behavior, such as excessively long running times or high memory consumption.

It's important to note that performance testing can be subjective to factors such as hardware, software environment, and implementation-specific optimizations. Therefore, it is recommended to perform the testing on multiple environments and analyze the results accordingly.

By combining functional testing to ensure the correctness of the outputs and

performance testing to evaluate the efficiency, we can gain confidence in the implementation's correctness and assess its behavior in different scenarios.

9. Discussion

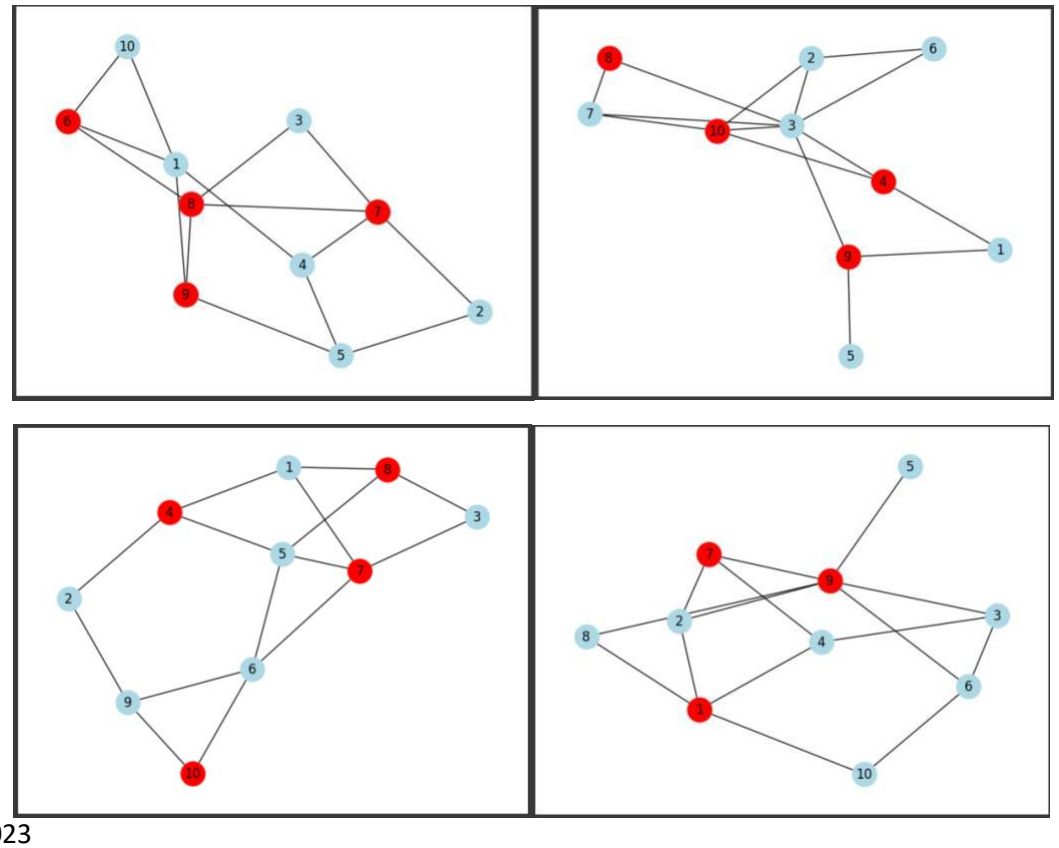
The greedy algorithm has an $O(n^3)$ time complexity, where n is the number of network nodes. This is due to the while loop iterating over all remaining undominated nodes and checking their neighbors for each iteration. This might not be effective for higher n and m valued graphs. To decrease runtime, think about utilizing more effective data structures or algorithms, like a priority queue or adjacency matrix.

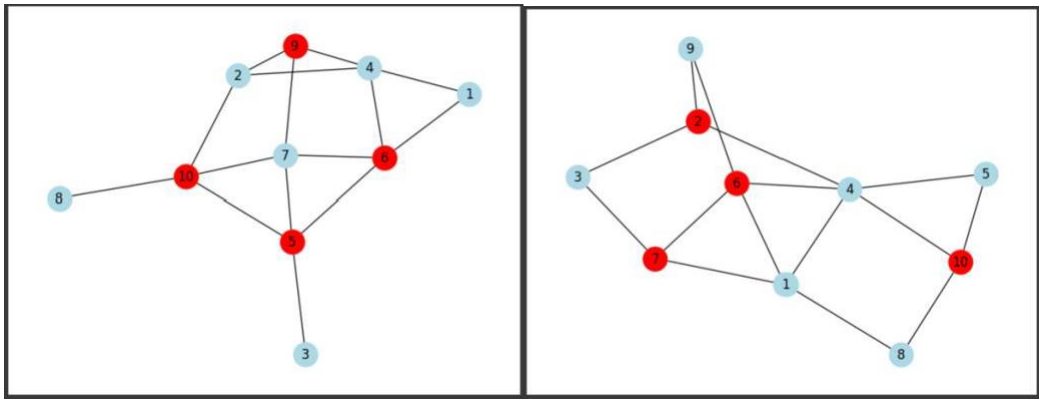
There seems to be no inconsistency observed for both algorithms in terms of theoretical and experimental analysis. Regarding the succession of finding the smallest dominating set, as it is previously stated, the greedy algorithm is not always successive.

For the time complexity of the algorithms, the experiments are observed to prove the O (n * (n + m)) Greedy Algorithm's and $O(2^n * n^2)$ Dominating Set Algorithm's time complexities.

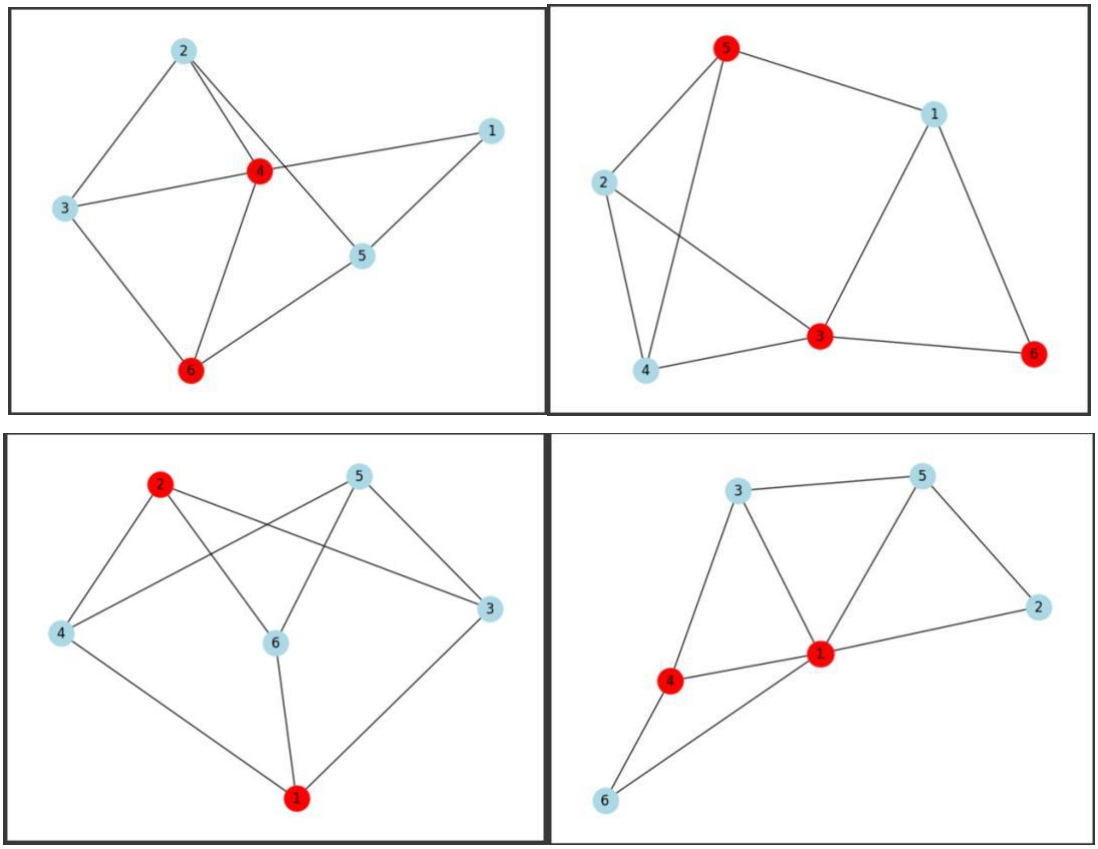
10. Appendices

Samples of n=10, m=15 (Brute Force Algorithm):

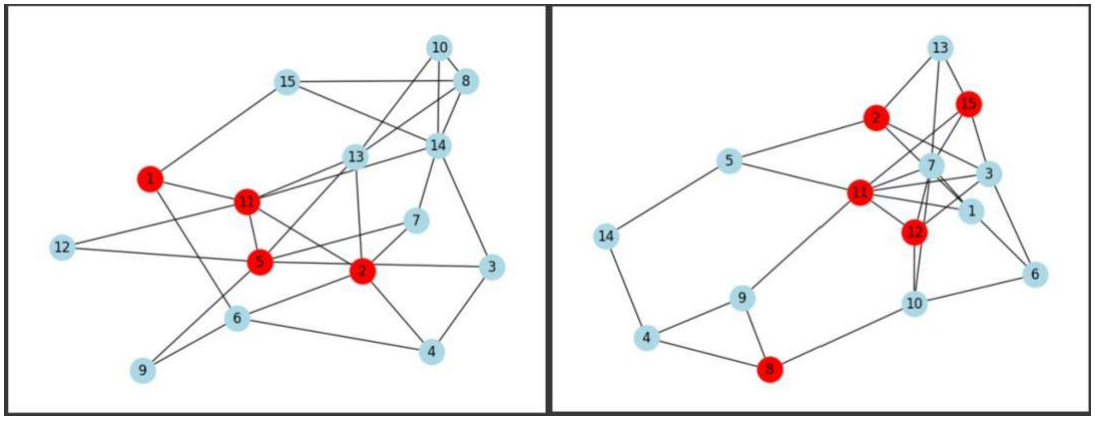




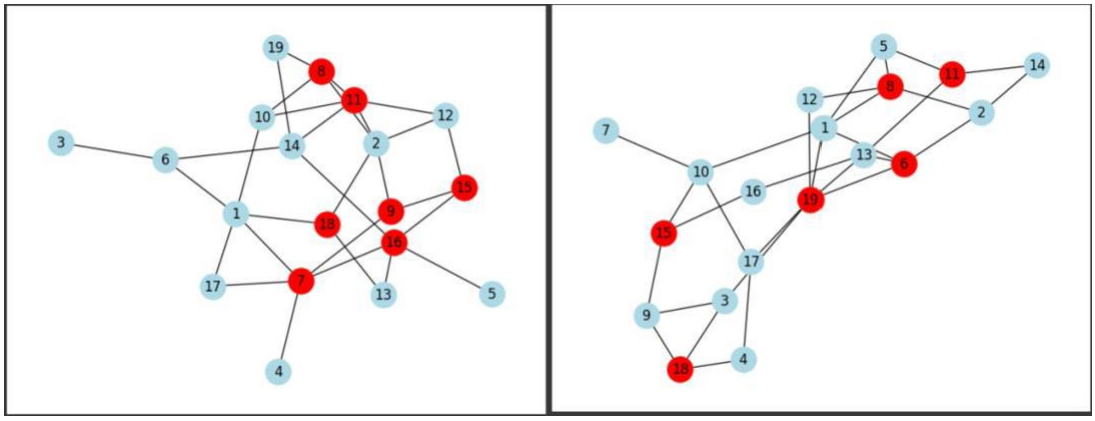
Samples of n=6, m=9 (Brute Force Algorithm):



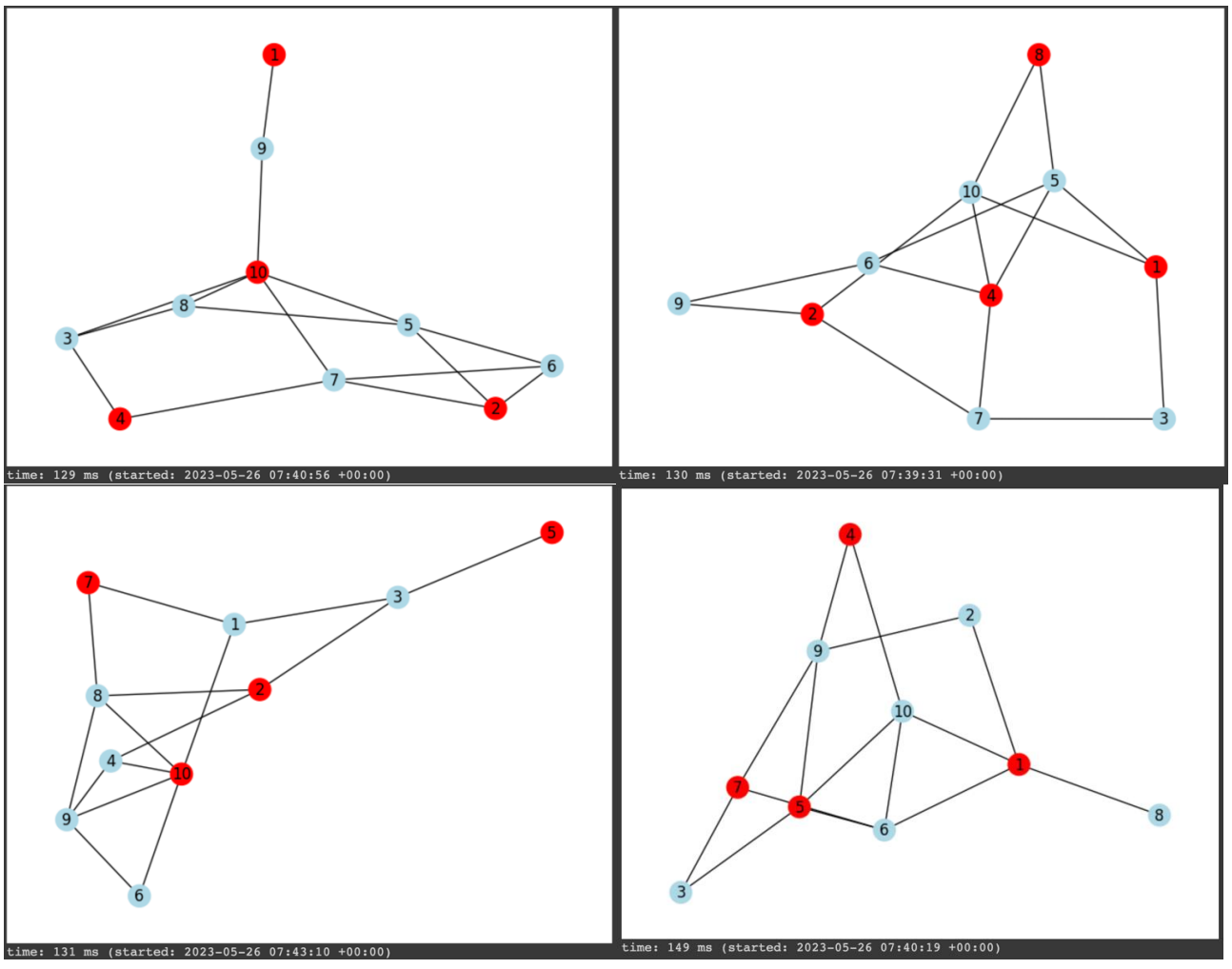
Samples of n=15, m=29 (Brute Force Algorithm):

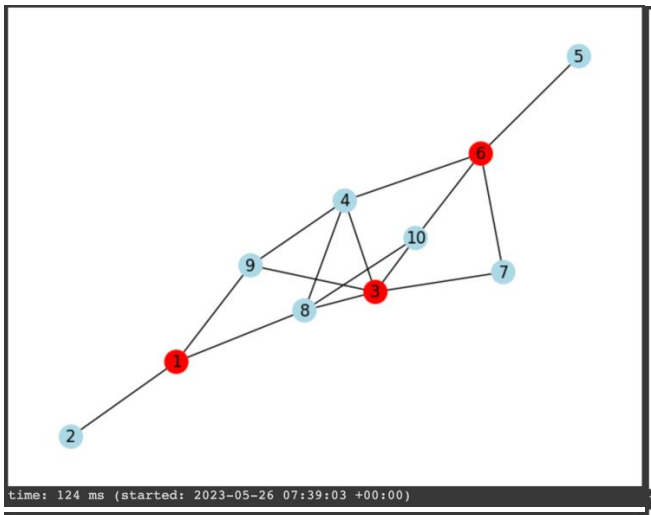


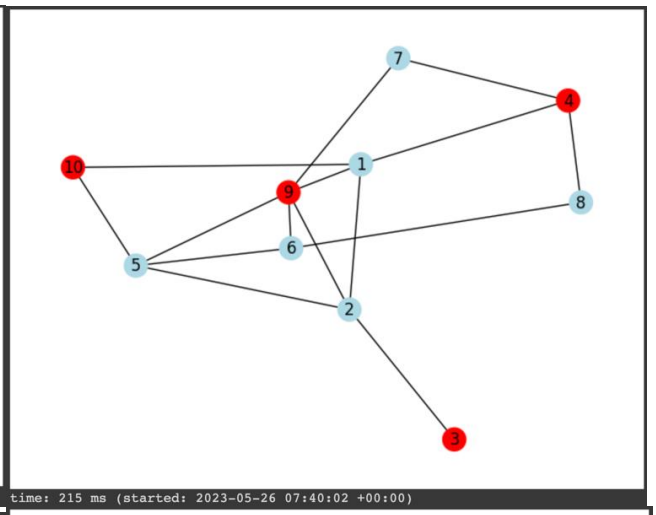
Samples of n=19,m=30 (Brute Force Algorithm):

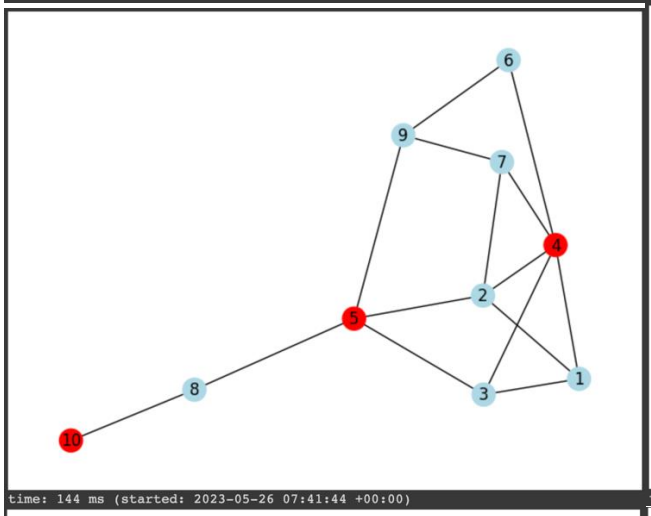


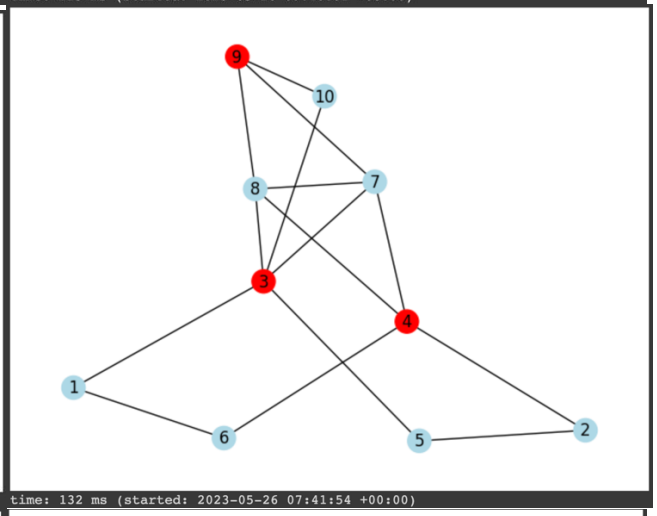
Samples of n=10, m=15 (Heuristic Algorithm):

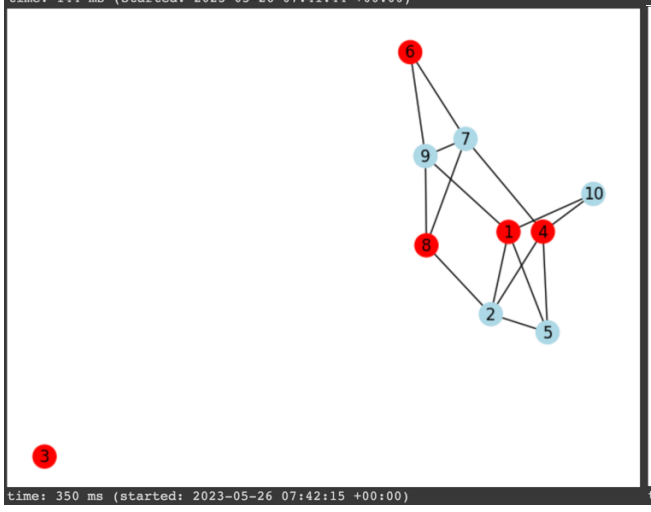


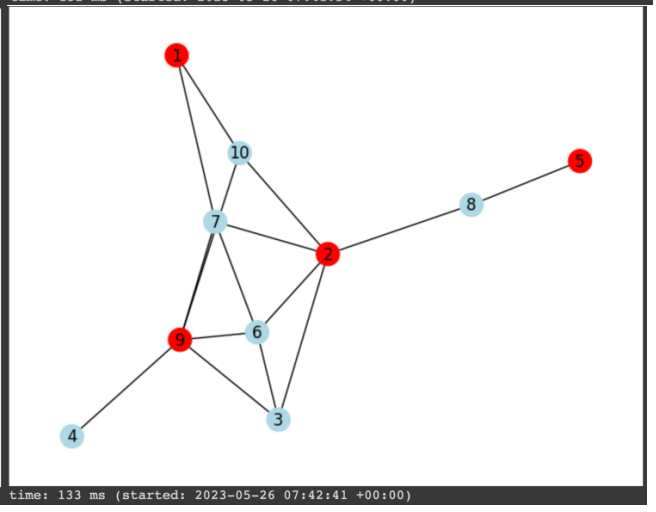

5
6
4
10
9
7
3
8
1
2
time: 124 ms (started: 2023-05-26 07:39:03 +00:00)

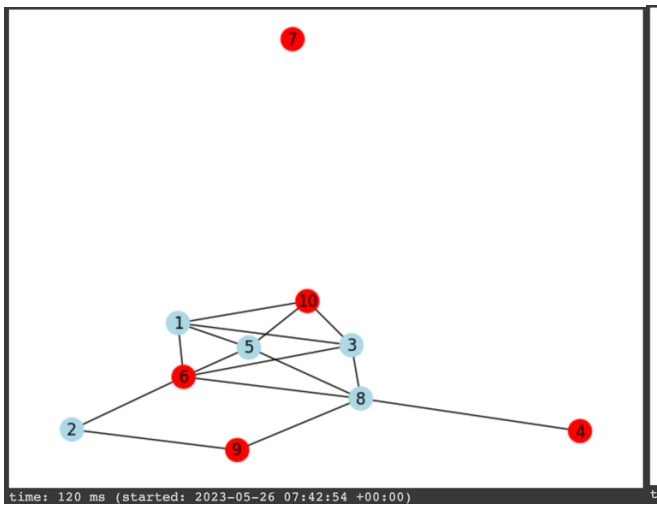

7
4
10
1
9
8
6
5
2
3
time: 215 ms (started: 2023-05-26 07:40:02 +00:00)

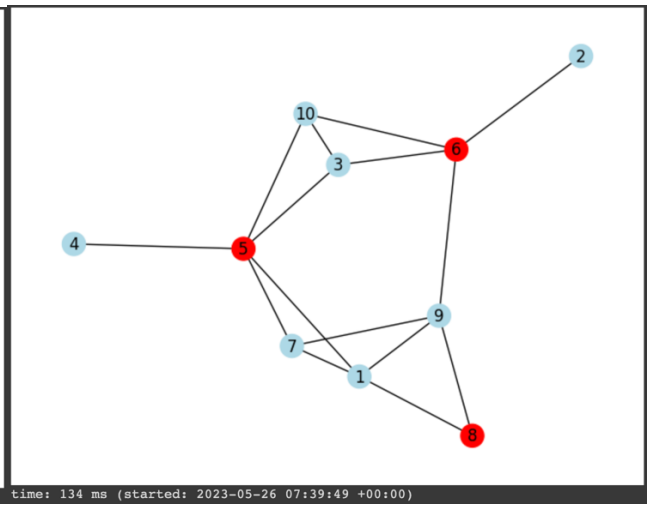

6
9
7
4
2
5
8
3
1
10
time: 144 ms (started: 2023-05-26 07:41:44 +00:00)


9
10
8
7
3
4
1
6
5
2
time: 132 ms (started: 2023-05-26 07:41:54 +00:00)

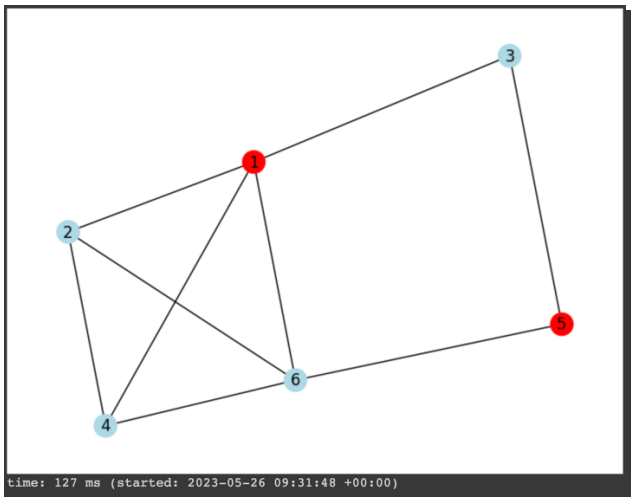

6
7
9
10
1
4
8
2
5
3
time: 350 ms (started: 2023-05-26 07:42:15 +00:00)

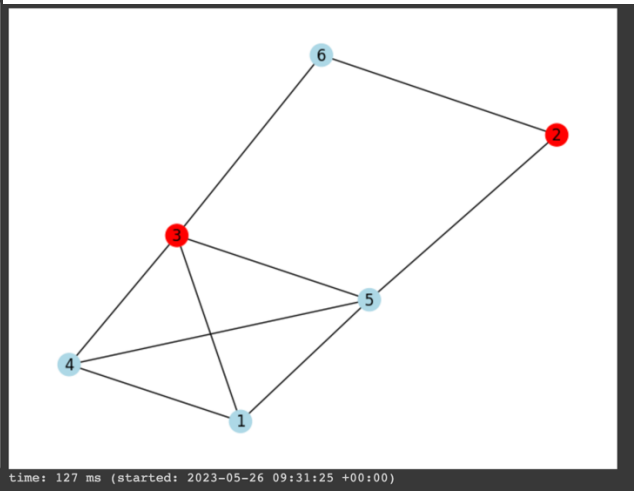

1
10
5
8
7
2
9
6
4
3
time: 133 ms (started: 2023-05-26 07:42:41 +00:00)



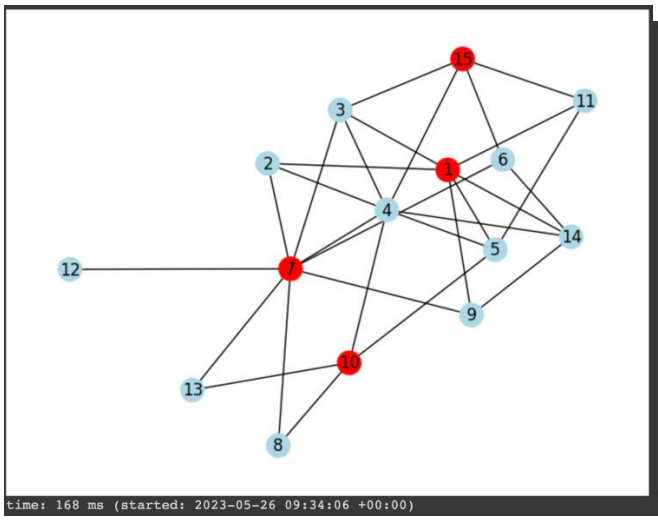


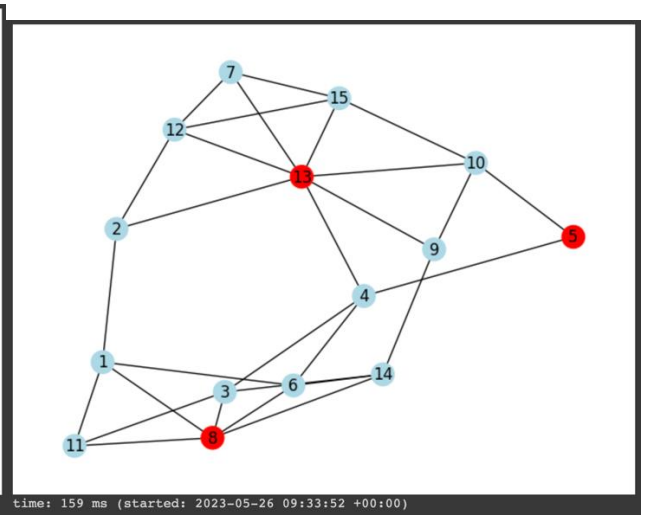
Samples of n=6, m=9 (Heuristic Algorithm):

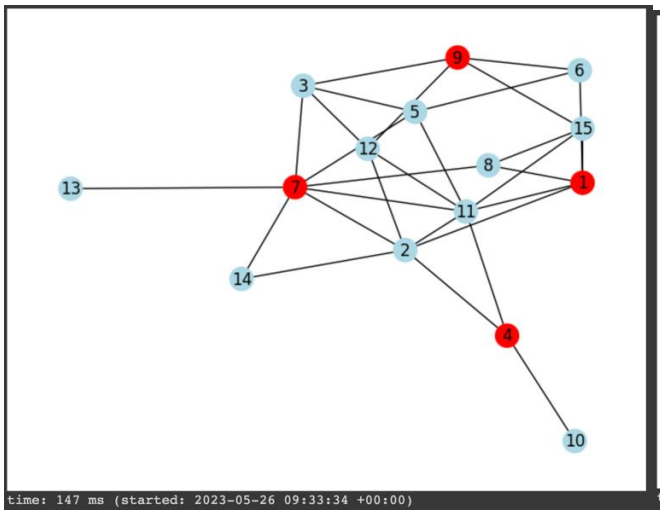


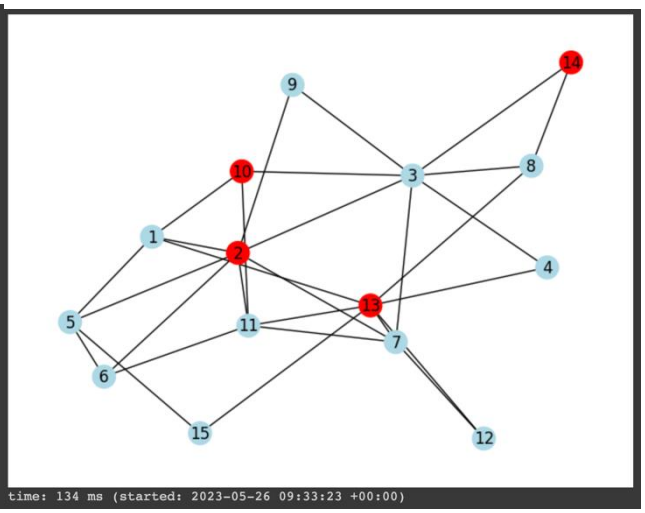


Samples of n=15, m=29 (Heuristic Algorithm):

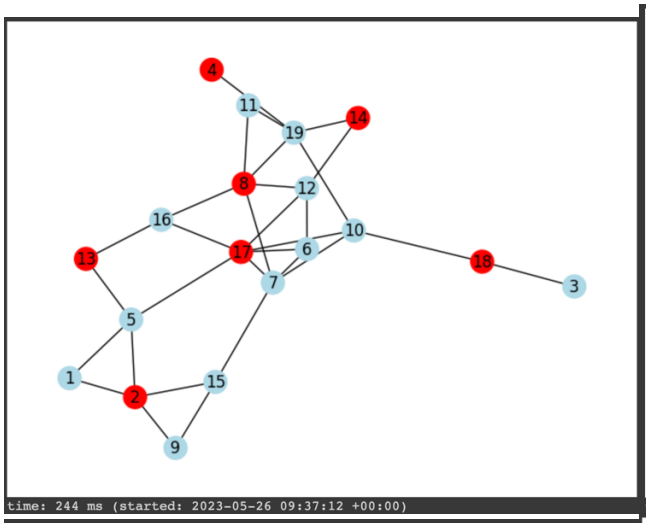


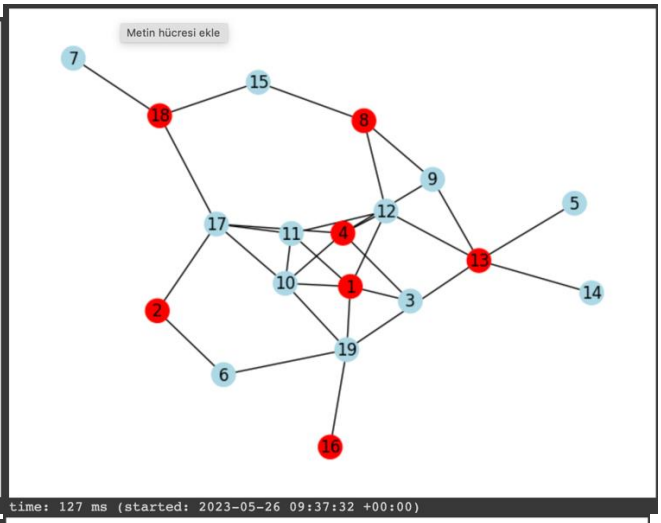


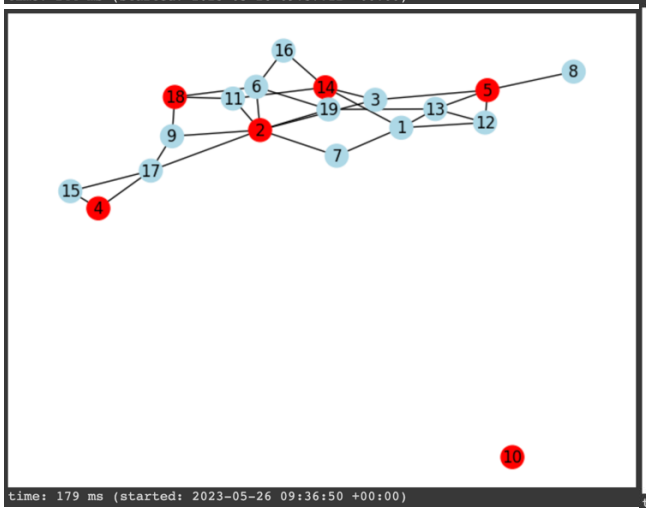


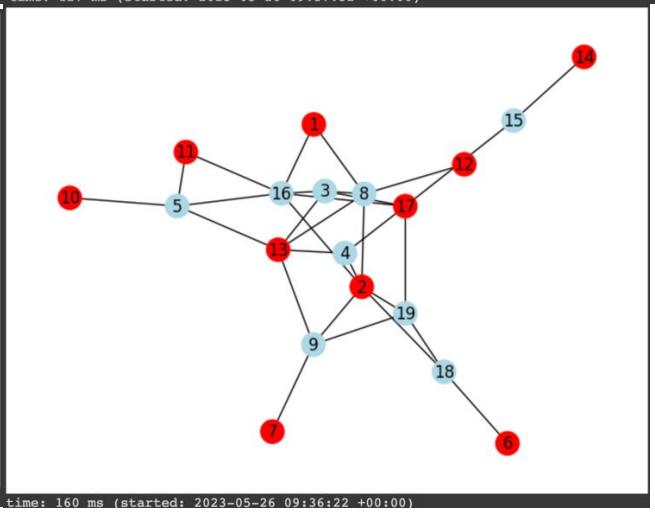


Samples of n=19, m=30 (Heuristic Algorithm):









Section 7: Algorithm Comparison Part:

Example of 50 times loop starting with n=6, m=9 and increasing n and m by 1 every 3 iteration

# Iteration	Brute Force	Greedy
0	3	1
1	2	3
2	2	2
3	2	1
4	3	3
5	3	3
6	3	2
7	3	3
8	3	3
9	3	2
10	4	3
11	3	5
12	4	4
13	3	5
14	4	3
15	4	4
16	3	5
17	4	5
18	4	6
19	4	5
20	5	6
21	4	6
22	5	5
23	5	5
24	5	5
25	5	6
26	4	7
27	5	6
28	5	6
29	5	7
30	4	7
31	6	7
32	6	7
33	6	6
34	6	9
35	6	9
36	6	7
37	6	9
38	7	9
39	7	9
40	7	8
41	6	9
42	6	10
43	7	10
44	7	10
45	7	9
46	7	10
47	8	10
48	7	12
49	8	11

```
Average time for BruteForceDominatingSet: 2.2391552686691285 seconds
Average time for greedyDS: 8.396625518798828e-05 seconds
Brute Force Total DS Nodes: 242
Greedy Total DS Nodes: 305
The Ratio of Greedy Total DS Nodes / Brute Force Total DS Nodes: 1.2603305785123966
time: 1min 51s (started: 2023-05-26 12:02:29 +00:00)
```

11. References

[1] Karp, Richard M. "Reducibility among combinatorial problems." In Complexity of computer computations, pp. 85-103. Springer, Boston, MA, 1972.