Algoritmos Avançados - Randomized Algorithm

Tiago Santos

***Abstract* –The objective of this assignment was to solve the problem of finding the maximum clique in a graph using a randomized algorithm. Tests were conducted to evaluate the effectiveness of this method.**

I. Introduction

This report explores the use of a **randomized algorithm** for determining the maximum clique of a graph. The maximum clique is defined as a subgraph where all its nodes are adjacent to each other, Figure 1. The focus is on evaluating the algorithm's efficiency by measuring its execution time and the success rate in accurately identifying maximum cliques.

II. Problem contextualization

The problem at hand revolves around the identification of the maximum clique within a provided graph and this task is approached through the utilization of a randomized algorithm. The implementation involves creating a graph using the Network [1] library, offering flexibility in constructing graphs either by specifying a fixed number of nodes and a percentage of edges or by reading the graph structure from an external file.

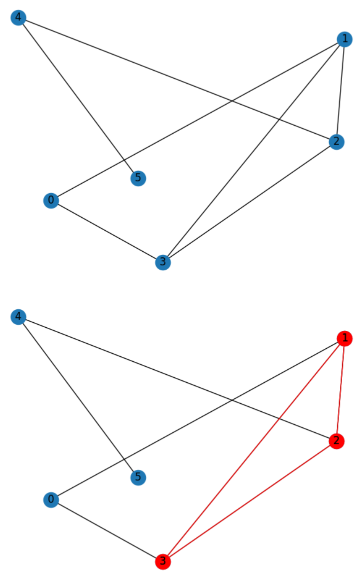

Fig. 1: Example of maximum clique

III. Implementation

A. Randomized Algorithm

A randomized algorithm [2] is an algorithm that uses random numbers to make decisions during its execution. This means that, unlike deterministic algorithms, which produce the same output for a given input every time they are run, randomized algorithms introduce an element of randomness into their decision-making process. This randomness can be introduced through the use of random numbers or probabilistic choices.

B. Code

For the context of this project, the function **randomized_max_clique** was created to solve the problem of maximum clique. This function leverages the principles of a Monte Carlo randomized algorithm to explore the solution space and probabilistically to find a maximum clique. The primary objective of the function is to approximate a solution that may not be optimal but is acceptable within a certain probability.

The algorithm initiates by obtaining a list of all nodes present in the given graph. Subsequently, it employs a randomized approach to form cliques by iterating through a specified number of tries. In each attempt, the algorithm randomly permutes the order of the nodes, essentially selecting a starting point at random. It then endeavors to grow a potential clique by iteratively adding nodes to the current set.

It then checks if a given set of nodes forms a clique within the graph using the is_clique function which verifies if each pair of nodes are connected by an edge. As it iterates through its randomized iterations, it maintains and updates **max_clique** variable to store the largest clique discovered.

It is important to note that due to the Monte Carlo nature of the algorithm, the result obtained may not always represent the true maximum clique. It is expected that the probability of success increases with the number of tries specified, providing a trade-off between computational efficiency and the accuracy of the approximation.

This process was created in the file main.py.

```
def randomized_max_clique(graph, num_tries):
    global op_count
    max_clique = set()

    for _ in range(num_tries):
        vertices = list(graph.nodes)
        random.shuffle(vertices)

        current_clique = set()
        for v in vertices:
            op_count += 1
            if is_clique(graph, current_clique | {v}):
                current_clique.add(v)

        if len(current_clique) > len(max_clique):
            max_clique = current_clique.copy()

    return max_clique
```

C. Formal Computational Complexity Analysis

Performing a formal computational analysis of the greedy search algorithm, regarding the time complexity:

$$O(N * logN * T)$$

Where **N** represents the number of nodes and **T** the number of attempts. The algorithm performance scales linearly with the number of vertices in the graph. As the number of vertices **N** increases, the time complexity increases proportionally. The number of attempts **T** also influences the overall time complexity, since increasing it improves the probability of finding the largest clique at the cost of increased computational time.

D. Usage

To run the code the user must first choose between using one existing graph or a randomly generated one by inputting the number of nodes and percentage of edge creation.
As such:

- Existing graph:

```
python3 main.py name_file.txt
```

- Random generated graph:

```
python3 main.py <number of nodes> <% of edge creation>
```

IV. Results and Discussion

Certain tests were made to better understand the algorithm's performance. This includes:

- **Average Time and Operation count** - Average Time and amount of operations by number of attempts;
- **Success Ratio** - Percentage of success by number of attempts;

A. Average Time and Operation Count

To test this, the file experiment.py was created which consists of the same process as the main.py but instead of specifying the number of attempts, it starts with some attempts equal to 1% of the number of nodes in the graph and increments it by 10% until it reaches around 90%.
For this test, it was used two files each one to form a different graph: SWtinyG.txt with 10 nodes and SWmediumG.txt with 250 nodes, and another randomly generated with around 500 nodes and 50% of percentage creation to give a different perspective in terms of number of nodes. For every iteration, it repeats the process 10 times, getting the time it takes to perform each attempt and the operation count.
For SWtinyG.txt:

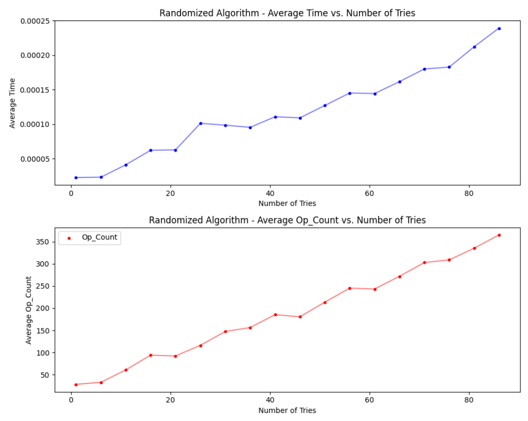

Fig. 2: Average Time and Operation Count vs Number of Attempts for tiny graph

For SWmediumG.txt:

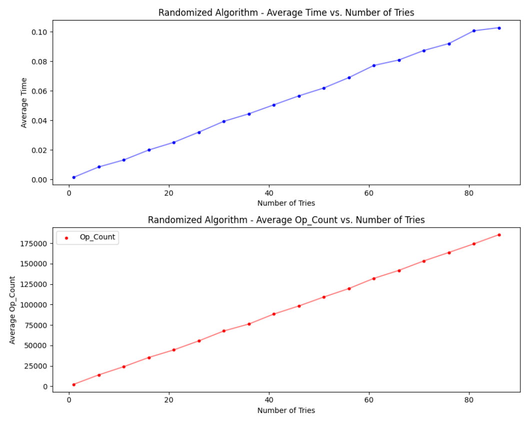

Fig. 3: Average Time and Operation Count vs Number of Attempts for medium graph

For random generated one:

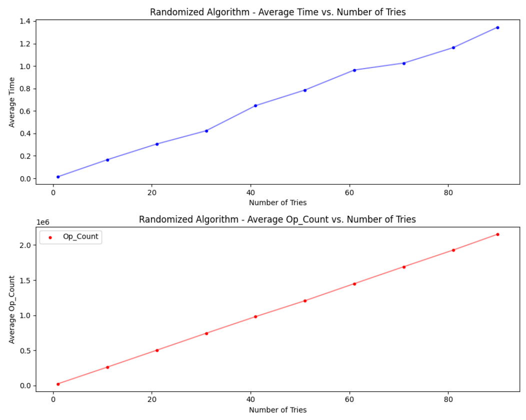

Fig. 4: Average Time and Operation Count vs Number of Attempts for randomly generated graph

As we can see from the results, both the Average Time and Operation count increase exponentially with the number of attempts, indicating a proportional rise in computational complexity as the algorithm iteratively explores larger sets of randomized trials.

B. *Success Ratio*

To perform it, it was used the same graphs utilized in the previous test, but in this case, we determine the percentage of success depending on the number of tries. We also utilize the same intervals of attempts. This was done in another Python file: experiment_succ.py

For SWtinyG.txt:

Number of attempts	Success Percentage (Run 1)	Success Percentage (Run 2)	Success Percentage (Run 3)	Average Success Percentage
1	10.0	70.0	40.0	40.0
2	60.0	80.0	40.0	60.0
3	80.0	60.0	50.0	63.33
5	90.0	100	100	96.66
6	100	90.0	90.0	93.33
7	90	100	100	96.66
8	100	90.0	100	96.66
10	100	100	100	100
11	100	100	100	100
12	100	100	100	100

For SWmediumG.txt:

Number of attempts	Success Percentage (Run 1)	Success Percentage (Run 2)	Success Percentage (Run 3)	Average Success Percentage
3	0.0	20.0	10.0	10.0
28	60.0	60.0	30.0	50.0
53	90.0	70.0	80.0	80.0
78	80.0	80.0	70.0	76.66
103	100	90.0	100	96.66
128	100	100	100	100
153	100	100	100	100
178	100	100	100	100
203	100	100	100	100
228	100	100	100	100

As we can see from the results in the previous tables, the lower the number of attempts the smaller the change of success. In this case the number of nodes doesn't make much of a difference in the results.

It is important to note that this test, like the previous one can be done either with a specific graph taken from a file or a randomly generated one.

V. CONCLUSION

After exploring the randomized algorithm, we can certain advantages of using as well as some disadvantages. Despite this being in a certain way, a faster algorithm its effectiveness can be limited due to the lack of certainty in its result, but with the right balance between the number of attempts and the number of nodes it can be an effective algorithm so solve not just the maximum clique problem but more problems that may require a fast search method.

REFERENCES

[1] NetworkX developers. NetworkX. `https://networkx.org/`

[2] Randomized Algorithm wiki. `https://en.wikipedia.org/wiki/Randomized_algorithm`