Randomized Algorithms for Combinatorial Problems – 2nd project

Daniel-Mihai Miricel

Abstract - **This report explores the implementation and analysis of a randomized approach to solving the clique problem in undirected graphs. The clique problem, which involves finding a complete subgraph of size k within a given graph, is computationally intensive and classified as NP-complete. The randomized approach will be compared to the previously analyzed approaches: the exhaustive search and the greedy heuristic.**

I. Introduction

The clique problem is a fundamental challenge in graph theory and combinatorial optimization. The problem entails identifying a complete subgraph, or clique, of a given size k within an undirected graph G(V,E). Despite its deceptively simple formulation, the problem is NP-complete [1] , meaning no polynomial-time algorithm is known to exist for solving it in the general case. This computational complexity makes the clique problem a perfect example for evaluating algorithmic strategies, particularly in balancing solution quality and computational efficiency.

This report focuses on solving the clique problem using a randomized approach and comparing it with the algorithms from the previous report. Since the clique problem is a decision problem, for the randomized approached I chose two Monte Carlo algorithms: an incremental algorithm and a sampling algorithm.

The exhaustive search approach, while computationally intensive, guarantees finding a solution by exploring all possible subsets of vertices. This makes it an ideal candidate for exact solutions on small graphs.

The greedy heuristic incrementally builds a clique by selecting vertices with the highest degree of connectivity, offering a computationally efficient alternative for larger or more complex graphs but without guaranteeing optimality.

The Monte Carlo incremental approach starts with an empty clique and iteratively adds random vertices to the clique if they maintain its properties. If a vertex doesn't fit, it's skipped, and another is randomly chosen. This approach relies on random exploration while gradually constructing the clique.

The Monte Carlo sampling algorithm will attempt to randomly sample subsets of vertices and check for cliques. If a clique of size k is found, it stops early; otherwise, it continues until a set number of iterations is completed.

The algorithms are tested on randomly generated graphs with varying characteristics, using both sparse and dense edge configurations, as well as on pre-defined graphs.

II. Monte Carlo sampling algorithm

The Monte Carlo sampling approach to solving the clique problem is a randomized algorithm that explores the solution space by selecting subsets of vertices at random. In each iteration, a set of k vertices is sampled from the graph uniformly at random. The sampled set is then tested to determine if it forms a clique, meaning all vertices in the set must be pairwise connected. If the sampled set satisfies the clique condition, the algorithm terminates successfully. Otherwise, the process continues until a pre-defined number of iterations is reached.

This method relies on the principle that, with enough random samples, the algorithm has a reasonable probability of identifying a valid clique if one exists. By tracking subsets that have already been tested, the algorithm avoids redundant computations, ensuring more efficient exploration. While it lacks the systematic nature of exhaustive search, the Monte Carlo sampling approach trades completeness for speed, making it particularly suitable for large and dense graphs where exhaustive methods are infeasible.

III. Monte Carlo incremental algorithm

The Monte Carlo incremental approach to solving the clique problem is a randomized algorithm that progressively builds a clique by adding vertices one at a time, starting from an empty set. The algorithm randomly selects a vertex from the graph and tests whether it can be added to the current clique. To qualify, the vertex must be connected to all existing members of the clique. If the vertex satisfies this condition, it is included; otherwise, it is skipped. This process continues until the clique reaches the target size k or all vertices are exhausted.

To improve its effectiveness, the algorithm goes through a pre-determined number of iterations. At the start of each iteration, the vertex set is shuffled. Doing multiple iterations enhances the probability of finding a clique by initiating independent attempts from different random starting points. In cases where a vertex fails to fit into the clique, the algorithm skips it without backtracking. This makes it less exhaustive compared to other methods. Despite this limitation, the Monte Carlo incremental

approach is well-suited for larger graphs where exhaustive search methods are not a possibility. By combining incremental construction, randomness, and multiple iterations, the algorithm provides a flexible and scalable way to approximate solutions to the clique problem.

IV. Formal Computational Complexity Analysis

This part of the report will show the formal time complexity analysis of the two algorithms. For this purpose I used the "big O notation" [2].

Each algorithm as a graph input of $G(V, E)$, where V is the number of vertices and E the number of edges. For each algorithm, the process is repeated T times, where T is a pre-defined fixed number.

A. Sampling algorithm

As mentioned earlier, the Monte Carlo sampling algorithm selects a subset of k vertices, then checks if it forms a clique. This process is repeated T times.

Random sampling k vertices has a complexity of $O(k)$.

Sorting the sampled subset adds a complexity of $O(k \log k)$.

For a subset of size k, the algorithm checks all combinations pairs of vertices to ensure they are connected.

$$\binom{k}{2}$$

The total time complexity will be:

$$O\left(T*\left(k + k\log k + \binom{k}{2}\right)\right)$$

B. Incremental algorithm

The Monte Carlo incremental algorithm randomly shuffles all vertices, builds a clique incrementally by adding vertices that are connected to all existing members of the clique, and stops once it reaches a clique size of k. The process is repeated T times.

Shuffling the vertex list requires a complexity of $O(V)$.

For each vertex, the algorithm checks whether it is adjacent to all vertices already in the clique. Checking for k members is $O(k)$. In the worst case, all V vertices are processed.

Total time complexity will be:

$$O(T*(V + V*k))$$

V. Experimental Computational Complexity Analysis

For the purpose of the experimental computational complexity analysis, I added a global counter into each of the algorithms, counting the total number of operations. In this phase I used graphs of increasing sizes, from 4 to 100 vertices, but with a fixed density of 50%. The pre-defined number of iterations chosen was T = 1000. The values resulted from the counting were plotted using matplotlib.pyplot [3].

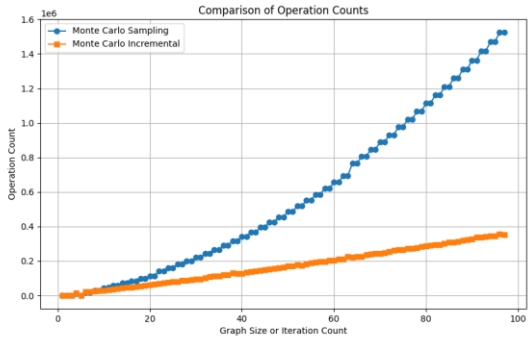

Figure 1 - comparison of operation counts, density 50%, up to 100 vertices

From the plot we can see how the Monte Carlo sampling algorithm increases much faster than the incremental algorithm.

It is important to note that this is not the case for all graphs and densities. For graphs of increasing sizes, from 4 to 300 vertices, with a fixed density of 10%, the Monte Carlo sampling algorithm is still less efficient than the incremental algorithm, however the difference becomes less obvious.

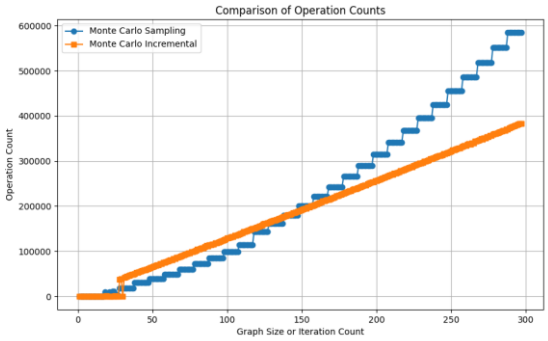

Figure 2 - comparison of operation counts, density 10%, up to 300 vertices

For graphs of increasing sizes, from 4 to 300 vertices, with a fixed density of 2%, the Monte Carlo sampling algorithm now becomes much more efficient than the incremental algorithm.

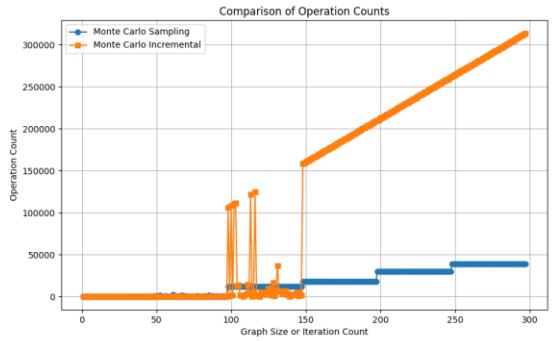

Figure 2 - comparison of operation counts, density 2%, up to 300 vertices

A. Sampling algorithm

In the case of the graph with 20 vertices, and a density of 50%, resulting in 95 edges, and a clique size of 10, I will compare the output of the experimental computational complexity analysis with the formal computational complexity analysis. With the experimental computational complexity analysis, the number of operations counted was 84892. With the formal computational complexity analysis we have the former formula, and the following variables:

$$V = 20, k = 10, E = 95, T = 1000$$
$$O(1000*(10 + 10 \log 10 + 10*9/2)) =$$
$$O(1000*(10 + 23 + 45) =$$
$$O(78000)$$

B. Incremental algorithm

Using the same graph that I did for the previous analysis, the number of operations counted was 52261. With the formal computational analysis we would get a result of:

$$O(1000*(20 + 20*10)) =$$
$$O(1000*(220) =$$
$$O(220000)$$

The big difference between the experimental analysis and the formal one can be explained by the way the incremental algorithm works. In the formal analysis, for each vertex in the shuffled list, the algorithm checks if it is adjacent to all the vertices in the current clique. This gives us a worst case scenario of $O(k)$ adjacency checks per vertex. However, in the experimental analysis, If vertices are shuffled in such a way that fewer adjacency checks are needed, for example if the first vertices are isolated, or if a clique is found early, the counter will track fewer operations than the formal analysis would suggest. In our case, since the graph has a density of 50%, it would be impossible for the algorithm to perform $O(k)$ adjacency checks per vertex every time.

VI. Precision of the Monte Carlo Algorithms

To analyze the efficiency of the Monte Carlo algorithms, I ran both algorithms and I compared them with the results from the exhaustive search algorithm, on the same set of graphs. To check how the number of iterations chosen for the Monte Carlo algorithms influences the precision, I chose three different values for the T number of iterations: 100, 1000 and 10000. The graphs were generated randomly, with a total number of vertices ranging from 4 to 25, densities of 12.5%, 25%, 50% and 75%, and the value k set to 12.5%, 25%, 50% and 75% of the number of graph vertices.

The total number of graphs generated was 352.

For $T=100$, the sampling algorithm had 20 false negative results, and the incremental algorithm had 0 false negative results. This resulted in a 94.3% precision for the sampling algorithm, and a 100% precision for the incremental algorithm.

For $T=1000$, the sampling algorithm had 4 false negative results, and the incremental algorithm had 0 false negative results. This resulted in a 98.9% precision for the sampling algorithm, and a 100% precision for the incremental algorithm.

For $T=10000$, the sampling algorithm had 3 false negative results, and the incremental algorithm had 0 false negative results. This resulted in a 99.1% precision for the sampling algorithm, and a 100% precision for the incremental algorithm.

For the same inputs, the greedy heuristic algorithm had a precision of 96.6%.

VII. Monte Carlo Algorithms on the Sedgewick & Wayne Graphs

For the current clique problem it makes sense to test the undirected graphs. There are 3 such graphs in the Sedgewick & Wayne graph repository: SWtinyG, SWmediumG and SWlargeG.

SWtinyG has 13 vertices and 13 edges. The k values chosen were 2, 4, 7 and 10. For $k=2$, the output values were “true”. For all other values the result was “false”. The total time it took to run the algorithms was 0.0 seconds.

SWmediumG has 250 vertices and 1273 edges. The k values chosen were 32, 63, 125 and 188. For all the values, and for both algorithms, the output was “false”. The total time it took to run the algorithms was 0.8 seconds.

SWlargeG has 1000000 vertices and 7586063 edges. The k values chosen were 125000, 250000, 500000 and 750000. For all the values, and for both algorithms, the

output was “false”. The total time it took to run the algorithms was 7107.7 seconds (1 hour 58 minutes 27 seconds).

VIII. Monte Carlo Algorithms on social networks

The Monte Carlo algorithms were also used for the graphs found in the Stanford large network dataset collection [4]. Because of the fact that for the medium and large graphs, the values chosen for k were quite large and no clique was found, I decided to set the value k to 0.1%, 1%, 12.5%, 25%, 50% and 75% of the number of graph vertices.

A. Social circles: Facebook

“Social circles: Facebook” is a dataset which consists of 'circles' (or 'friends lists') from Facebook [5]. The graph consists of 4039 vertices and 88234 edges. The k-values chosen were 5, 41, 505, 1010, 2020 and 3030. Along with the two Monte Carlo algorithms I also tested the greedy heuristic algorithm for this graph. For k=5 the results were “true” for the greedy heuristic and for the Monte Carlo incremental, and “false” for the Monte Carlo sampling. For k=41 the result was “true” for the Monte Carlo incremental, and “false” for the greedy heuristic and the Monte Carlo sampling. For all other k values the results were all “false”. Total running time was 14.5 seconds. When ran one by one, the Monte Carlo sampling algorithm took 2.6 seconds, the Monte Carlo incremental algorithm took 13.1 seconds and the greedy heuristic algorithm took 0.1 seconds.

B. Deezer Romania

The data was collected from the music streaming service Deezer. These datasets represent friendship networks of users from 3 European countries [6]. Because of my nationality I chose the dataset from Romania. The graph consists of 41773 vertices and 125826 edges. The k-values chosen were 42, 418, 5222, 10444, 20887 and 31330. Along with the two Monte Carlo algorithms I also tested the greedy heuristic algorithm for this graph. For all of the values, and all of the algorithms, the results were all “false”. Total running time was 274.8 seconds. When ran one by one, the Monte Carlo sampling algorithm took 4 seconds, the Monte Carlo incremental algorithm took 230.5 seconds and the greedy heuristic algorithm took 26.3 seconds.

C. Last FM Asia

A social network of LastFM users which was collected from the public API in March 2020. Vertices are LastFM users from Asian countries and edges are mutual follower relationships between them [7]. The graph consists of 7624 vertices and 27806 edges. The k-values chosen were 8, 77, 953, 1906, 3812 and 5718. Along with the two Monte Carlo algorithms I also tested the greedy heuristic algorithm for this graph. For k=8 the results were “true” for the greedy heuristic and for the Monte Carlo incremental, and “false” for the Monte Carlo sampling. For all of the other values, and all of the algorithms, the results were all “false”. Total running time was 31.2 seconds. When ran one by one, the Monte Carlo sampling algorithm took 4 seconds, the Monte Carlo incremental algorithm took 29.8 seconds and the greedy heuristic algorithm took 0.1 seconds.

IX. Conclusions

After running both Monte Carlo algorithms on a various number of graph instances, with varying sizes and densities, and comparing them with the greedy heuristic and the exhaustive search algorithms, it became evident that the Monte Carlo methods offer a trade-off between accuracy and computational efficiency.

The Monte Carlo sampling algorithm had a lower computational cost for sparse graphs. However, its performance declined for denser graphs. The precision was the lowest out of the 4 algorithms.

The Monte Carlo incremental algorithm showed consistent performance across varying graph densities. By progressively building cliques and leveraging randomization, it managed to find solutions faster. Its incremental nature made it computationally way less expensive compared to exhaustive search. For high density graphs it proved to be faster than the sampling algorithm. It is worth noting that, for all the graphs where the exhaustive search was possible, the Monte Carlo incremental algorithm had a precision of 100%.

The greedy heuristic struggled with precision in graphs where high-degree vertices did not necessarily form cliques. Nevertheless it proved to be more precise than the Monte Carlo sampling algorithm for all the graphs it was tested on. It exceled in computational speed, making it suitable for very large graphs, but its reliance on vertex degrees limited its effectiveness in certain scenarios.

The exhaustive search algorithm, as expected, guaranteed correctness but proved computationally prohibitive for larger graphs. Its exponential growth in complexity rendered it impractical for most real-world scenarios.

In summary, the Monte Carlo algorithms provide a flexible middle ground between speed and accuracy. The choice between them depends on the graph's properties and the application's tolerance for approximate solutions.

ACKNOWLEDGEMENT

All code was written in Python3. All code was developed using PyCharm IDE [8]. All code was ran on my personal computer, with the processor AMD Ryzen™ 5 5600H. The graphs were generated, processed and manipulated using the NetworkX library [9]. Random values generated using Python's Random library [10]. Math library [11] used for Euclidean distance and for log function. Itertools library [12] used for combinations. Time library [13] used for counting runtime. Pandas library [14] used for reading csv files.

REFERENCES

[1]Proof that Clique Decision problem is NP-Complete, *Geeksforgeeks*. URL: https://www.geeksforgeeks.org/proof-that-clique-decision-problem-is-np-complete/

[2]Big O notation, *Wikipedia*. URL: https://en.wikipedia.org/wiki/Big_O_notation

[3]J. D. Hunter, "Matplotlib: A 2D Graphics Environment," in Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, May-June 2007, doi: 10.1109/MCSE.2007.55.

[4]Jure Leskovec, & Andrej Krevl. (2014). SNAP Datasets: Stanford Large Network Dataset Collection.

[5]J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

[6]B. Rozemberczki, R. Davies, R. Sarkar and C. Sutton. GEMSEC: Graph Embedding with Self Clustering. 2018.

[7]B. Rozemberczki and R. Sarkar. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. 2020.

[8]JetBrains, PyCharm URL: https://www.jetbrains.com/help/pycharm/2022.2/quick-start-guide.html

[9]NetworkX – Network analysis in Python URL: https://networkx.org/

[10] Random - Generate pseudo-random numbers

URL: https://docs.python.org/3.11/library/random.html

[11]Math - Mathematical functions

URL: https://docs.python.org/3.11/library/math.html

[12]Itertools - Functions creating iterators for efficient looping URL: https://docs.python.org/3.11/library/itertools.html

[13]Time - Time access and conversions

URL: https://docs.python.org/3.11/library/time.html

[14] Pandas URL: https://pandas.pydata.org/