Randomized Search for finding the Maximum Independent Set of a Graph

Miguel Matos, 103341

***Abstract* – Design, testing, and analysis of two computational algorithms to find the maximum independent set for a given undirected graph.**

I. Introduction

The field of combinatorial optimization encompasses a vast array of problems where the goal is to find an optimal object from a finite set of objects. Among these, the problem of identifying the maximum independent set in a graph is particularly noteworthy due to its wide-ranging applications in fields such as network theory, scheduling, and resource allocation. This report is centred on the development and analysis of a randomized algorithm to solve this specific problem.

The significance of the maximum independent set problem lies in its inherent complexity and the challenge it poses to traditional deterministic algorithms, especially in large and complex graph structures. This complexity has driven the exploration of alternative approaches, including randomized algorithms, which offer a different paradigm for problem-solving. Unlike deterministic algorithms, which follow a fixed sequence of steps, randomized algorithms incorporate elements of randomness, providing a means to explore the solution space in a more diverse and potentially more efficient manner.

This assignment aims to design, implement, and evaluate a randomized algorithm for finding the maximum independent set. The focus is not only on the algorithm's ability to find an optimal or near-optimal solution but also on its efficiency, scalability, and reliability in comparison to deterministic methods. The project includes a thorough computational complexity analysis and a series of experiments using various graph instances, both from the first project and additional benchmark graphs available on the web.

II. Randomized Search Algorithm

A. Design

A randomized search algorithm incorporates an element of randomness in order to influence the path of logic. Unlike deterministic algorithms, which produce the same output for a given input every time (for example, the exhaustive and greedy search methods studied in the first assignment), randomized algorithms can yield different outcomes on the same input on different executions.

The most basic randomized algorithm to find the maximum independent set of a given graph instance would test random subsets of graph nodes, checking if they are independent. After a given time or number of iterations, it would stop, returning the largest set found.

However, the performance of this algorithm can be improved, as to “tame” the randomness of the algorithm. Given the domain of the problem, we can limit the choices the algorithm makes along its path of logic to better guide it along the way.

Since the goal is to find the maximum size of any independent subset, one does not need to test subsets of size smaller or equal than the maximum set already found. This reduces the size of possible solutions as larger solutions are found, as well as we get closer to an optimal solution.

Also, the algorithm might end up generating a graph instance that was considered and discarded, so one can store the information about which subsets have already been considered to avoid wasting more compute resources on them. To efficiently store this information, a bit vector representation of the subsets was used, since its memory footprint would be much smaller. As the size of the graph increases linearly, the number of possible solutions increases exponentially, given that for n vertices, there are 2^n candidate subsets.

To further enhance the performance of the algorithm, another optimization was added. As with the greedy search, we know that nodes with lower degree have a higher chance of belonging to an independent set, since they are less likely to have neighbours in the generated subset. Thus, we weight the vertices, with value inversely proportional to their respective degrees, and generate a random sample of these using these weights. Weight normalization is also employed.

```
Create node_index_map as a mapping from each
node to its index in the graph
Initialize tested_bitvectors as an empty set
Initialize best_solution as an empty set
Set best_score to 0
Set failure_count to 0
For each iteration up to max_iterations:
  Generate a random choice of nodes, the
candidate solution, given:
    min_size > best_score
    node_weight = 1/node_degree
  Convert the candidate solution to a
bitvector using the node_index_map
  If the candidate bitvector is not in
tested_bitvectors:
    Increment tested_count by 1
    Add the candidate bitvector to
tested_bitvectors
    If candidate is an independent set:
       Calculate the score as the length of
the candidate
       Update best_solution with the
candidate
       Update best_score with the new score
       Reset failure_count to 0
     Else:
       Increment failure_count
       If failure_count is greater than or
equal to failure_threshold:
         Exit for
```

B. Formal Analysis

The computational complexity of the algorithm is mostly influenced by the generation of the candidate, and by the process of checking candidate for independence.

The generation of the candidate subset involves iterating over all nodes, calculating degrees and normalizing weights, which are dependent on the graph's edges. Thus, it's complexity is $O(n + m)$, where n is the number of vertices and m is the number of edges.

When checking the candidate for independence, each node in the subset leads to an iteration over its neighbours, proportional to the node's degree. This leads to a complexity of $O(kd)$, where k is the size of the subset and d is the average degree of nodes.

Since these components are combined in the main iteration loop of the algorithm, this can lead to a worst case scenario of $O(nm)$. This is because for each node (up to n), you might need to examine a number of edges proportional to the total number of edges m.

However, for sparse graphs, the complexity could be lower, since the number of edges is much smaller relative to the number of vertices.

C. Experimental Analysis

In order to corroborate the formal analysis results, the algorithm was run on randomly generated graphs with the same characteristics as the ones from the last assignment, as well as some benchmark instances (six Twitch related graphs and one Wikipedia related).

Four metrics were evaluated: execution time, number of basic operations performed, the number of combinations tested, and the size of the maximum independent set found. Results were obtained from averages of five runs of the algorithm.

I also decided to benchmark the final version (refered to as version 2) against a version of the algorithm without the node weighting during the candidate subset generation process (refered to as version 1), in order to also evaluate the impact that single change had in the algorithm.

The following settings were used for the experiments:

- max_iterations was set to 100 000
- timeout was set to 60 seconds
- failure_threshold was set to 100

a) Count of Operations

Both versions showed a steep increase in operation counts with larger graphs. The exponential growth pattern aligns with the expected complexity of combinatorial graph problems.

V2 consistently had higher operation counts than V1. This observation suggests that node weighting in V2 leads to more computational work per vertex, due to a more complex decision-making process in selecting nodes for the independent set.

However, V2 displays much more irregular and lower values at lower edge percentages, when comparing to higher edge percentages of the same version, while V1 provides results similar for all edge percentage values. This indicates that the node weighting process is more effective for sparser graphs, since sparser graphs will have a higher frequency of nodes with lower degrees, since there are less overall edges.

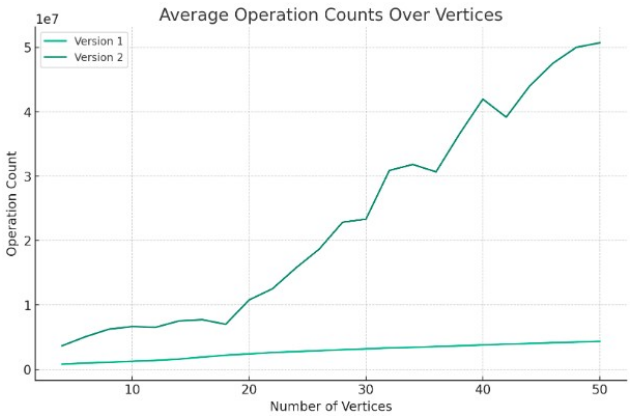

Figure 1- Operation Count Comparison between Versions

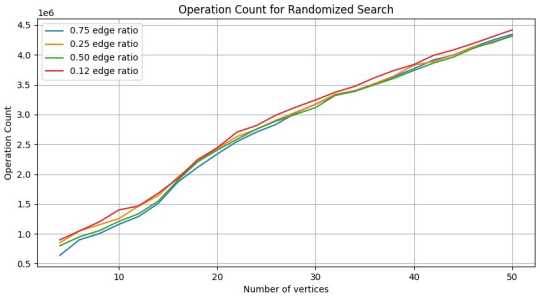

Figure 2- Operation Count For Version 1

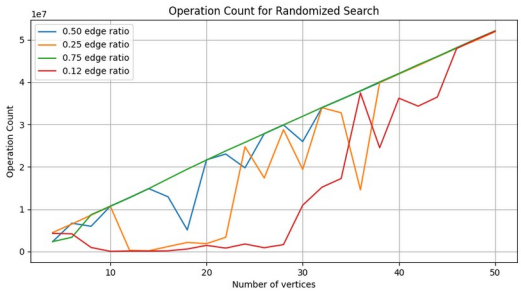

Figure 3- Operation Count for Version 2

b) Time Taken

V1 demonstrated a consistent increase in execution time as the number of vertices increased. This trend is typical for combinatorial problems, where the complexity escalates with the size of the graph.

V2, while following a similar upward trend, exhibited higher execution times at each vertex level compared to V1. The additional processing overhead in V2, likely due to the implementation of node weighting, accounts for this increased time complexity.

As with the operation counts, V2 exhibits much different behaviour for each edge percentage line when compared with V1. For a certain range of vertices (around [10, 20]), V2's times are even better than V1.

With this insight, we can make a prediction for execution time of this algorithm for much larger graph instances. As seen in Figure 2, for 20 vertices, at 50% edge ration, the algorithm takes around 1 second to execute. Assuming that the execution time grows by 1000 times for each jump of 10 vertices, a graph with 50 vertices with the same edge ratio should take around 10^{16} seconds, which equates to around 316 887 646 years. This is a lot.

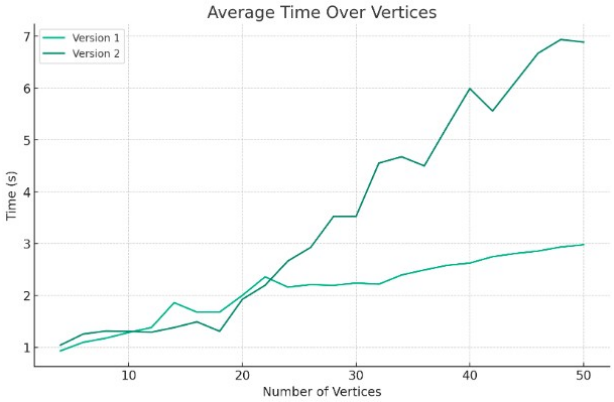

Figure 4- Time Comparison between Versions

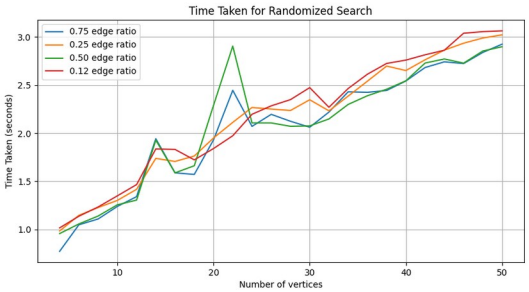

Figure 5- Times for Version 1

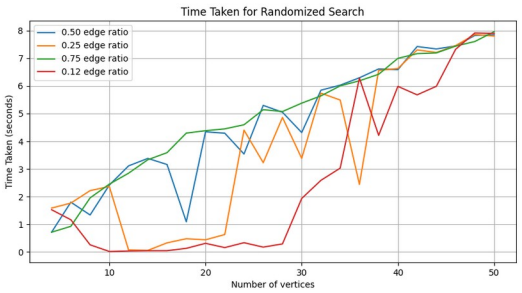

Figure 6- Times for Version 2

c) Number of Combinations Tested

There was a significant increase in the number of combinations tested as the graph size grew. This rise is expected due to the combinatorial nature of the problem, where possible subsets of vertices increase dramatically with graph size.

V2 tested fewer combinations than V1, especially in larger graphs. This reduction could be a result of the node weighting strategy in V2, which may lead to quicker convergence on promising solutions, thereby reducing the need to test as many combinations.

Like other parameters, and for the same reasons, V2's way of choosing the new candidate subset has higher benefits for lower edge percentages/sparser graphs.

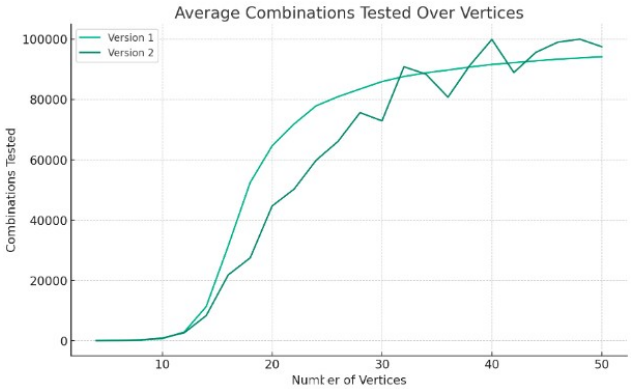

Figure 7- Combinations Tested Comparison between Versions

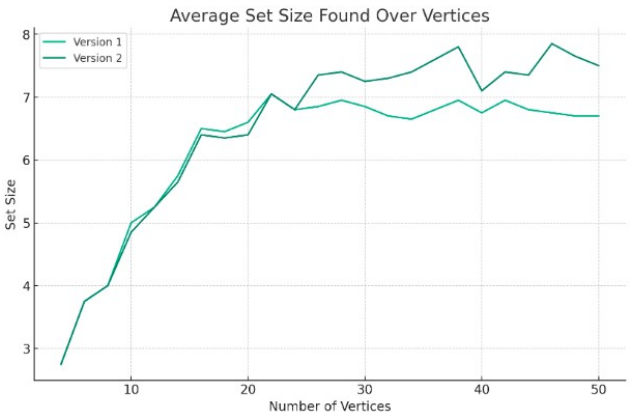

Figure 10- Set Size Found Comparison between Versions

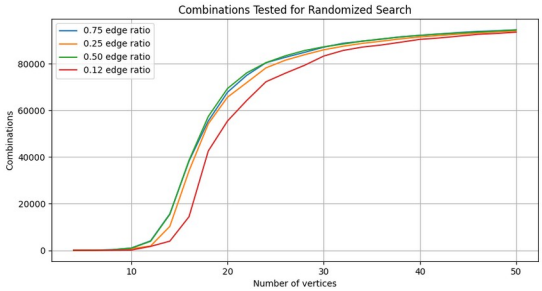

Figure 8- Combinations Tested for Version 1

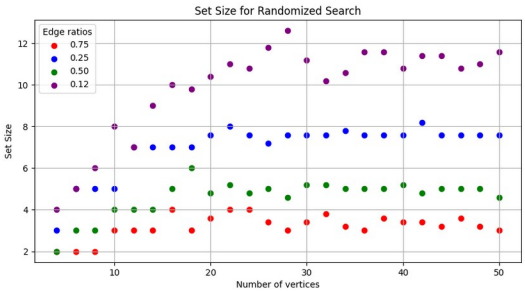

Figure 11- Set Size Found for Version 1

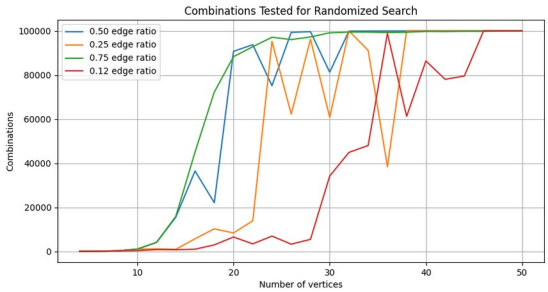

Figure 9- Combinations Tested for Version 2

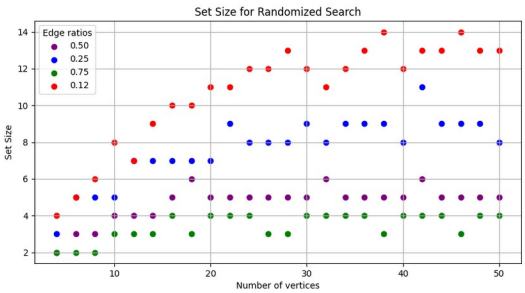

Figure 12- Set Size Found for Version 2

d) Set Size Found

The average set sizes found by both algorithms increased with the number of vertices. This increase is logical, as larger graphs naturally allow for larger independent sets.

The set sizes in V2 were marginally larger or equal to those in V1 across most vertex counts. This outcome indicates that the node weighting in V2 potentially contributes to finding slightly larger independent sets.

e) Benchmark instances

The algorithm was also tested on some larger graph instances, available on the web, for benchmarking purposes. The graphs used were:

Wikipedia Vote Network[6]

Graph	Vertices	Edges	Edge Percent
Wiki	7115	100762	0.398%

Twitch Social Networks[5]

Graph	Vertices	Edges	Edge Percent
DE	9498	153138	0.340%
ENGB	7126	35324	0.139%
ES	4648	59382	0.550%
FR	6549	112666	0.525%
PTBR	1912	31299	1.713%
RU	4385	37304	0.388%

Below are the results for each graph:

Graph	Time	Operations	Combinations	Set Size
Wiki	600	3563142451	49922	770
DE	600	2409209792	25343	290
ENGB	600	2904300331	40726	299
ES	600	2378418261	51095	230
FR	600	2377989238	36272	251
PTBR	245	1920031540	100000	134
RU	600	3759712784	85613	252

As we can see, the algorithm performs better on graphs with less vertices, due to the combinatorial nature of the problem. The only run that did not stop due to the time limit of 10 minutes (600 seconds) was the smallest one, which ended up stopping by the number of iterations, having tested a new subset of vertices each iteration.

IV. FINAL CONSIDERATIONS

In this report, a randomized algorithm for finding the maximum independent set for a given graph instance. Two versions of this algorithm were showcased, tested, and benchmarked. The difference between these two versions was on the process of randomly choosing a new candidate subset of nodes to test; in the first version, it is done completely at random, valuing each node as high as each other, but the second version weights nodes based on their degree, assuming that nodes with a lower degree are less likely to be the cause for dependence within the set, since these have less connecting edges.

The introduction of node weighting in V2, while increasing the computational load and execution time, has shown a propensity to yield slightly larger independent sets. This outcome underscores a fundamental trade-off in algorithm design: a more intricate and computationally demanding approach can potentially lead to better solutions, albeit at the cost of efficiency.

Another factor worth mentioning is that due to time constraints there was no attempt at tuning factors such as the number of maximum iterations, maximum run time allowed and failure threshold. For different graphs, the same value of these variables may impact the algorithm in a different way. As an example, for a denser graph, the algorithm would be more likely to stop due to reaching the failure threshold, since it would be more likely that the sets it was testing were not independent, and thus did not contribute to a better solution.

REFERENCES

[1] https://chat.openai.com/
[2] https://en.wikipedia.org/wiki/Optimization_problem
[3] https://en.wikipedia.org/wiki/Independent_set_(graph_theory)
[4] https://en.wikipedia.org/wiki/Greedy_algorithm
[5] https://snap.stanford.edu/data/twitch-social-networks.html
[6] https://snap.stanford.edu/data/wiki-Vote.html