AA - Maximum Weight Independent Set Randomized Algorithms

Alexandre Ribeiro - 108122

***Abstract* – In my previous report, I analyzed exhaustive and greedy algorithms for the Maximum Weighted Independent Set (MWIS) problem, finding that heuristic-based approaches could deliver near-optimal solutions efficiently. The most effective heuristic, based on the ratio of node weight to degree, achieved solutions close to the maximum weight. However, it was only perfectly accurate about 50% of the time for sparse graphs, with accuracy dropping to 18% as graph density increased. In this report, I present and evaluate three new algorithms that incorporate randomness, two of them inspired by the Monte Carlo method. These algorithms strike a promising balance between accuracy and computational efficiency, demonstrating significant improvements over greedy heuristics, especially for denser graphs. The results highlight their potential as a robust alternative for tackling MWIS problems in practical scenarios.**

***Keywords* – Maximum weighted independent set, Monte Carlo, randomized algorithm, parallel algorithm**

I. Introduction

The Maximum Weighted Independent Set (MWIS) is one of the most important problems in graph algorithms and has various applications in areas such as social networks [1] and computer graphics [2], which is why it is relevant to find better and better algorithms to solve this problem.

In the previous report, I explored exhaustive and greedy algorithms for solving the MWIS problem on increasingly larger graphs. The exhaustive method guarantees the optimal solution for a given graph $G(V, E)$, but the execution time grows exponentially and as such is limited to smaller graphs. The greedy method offers lighting-fast execution and achieves near-optimal solutions in many cases, depending on the heuristic, but overall fails when it comes to perfect accuracy, dropping to approximately 18% for denser graphs. As such, there is a need to find an algorithm that strikes the balance between being fast and being accurate.

The next step in this work is to develop algorithms that apply randomness. Randomized algorithms are appealing for their ability to explore the solution space efficiently while balancing accuracy and speed. [3] By introducing stochastic elements, they can avoid the local optima traps of greedy methods and offer a practical alternative to exhaustive searches for large graphs.

For the MWIS problem, randomness can help prioritize higher-value vertices. This could involve assigning random priorities based on heuristics and iteratively refining the solution. These methods offer a flexible trade-off between runtime and accuracy, making them ideal for large, complex graphs.

After researching the topic, I found that while there are many articles about randomized algorithms for the Maximum Independent Set problem, the same cannot be said for the weighted variant. That being said, I studied these articles in search of methods that could be applicable to the problem at hand.

The most common type of algorithm that I found revolved around the Monte Carlo algorithm, which relies on repeated random sampling to reach approximate solutions. Some variations developed by previous authors explored concepts such as d-wise independence, used to separate sections of the graph and solve them independently in order to improve solving times [4], or simulated annealing, which is a complex method that explores the solution space using a decreasing temperature [5]. That said, while these variations are both intriguing and highly applicable, fully implementing them as described by their authors would require significant time. However, the underlying concepts proved to be highly valuable and were incorporated into the development of the algorithms presented in this report.

In the end, I decided on three algorithms:

- **Monte Carlo (MC):** Simple implementation of the Monte Carlo algorithm, starts each iteration by randomly sorting the nodes and then reaching a solution locally, selecting the head node and removing its neighbors until there are no remaining nodes.
- **Parallel Heuristic Monte Carlo (PHMC):** Similar to the Monte Carlo, with a change to the sorting mechanism by attributing a priority to each node according to the Weight to Degree heuristic, the most promising heuristic as determined by the last report. This priority helps the

selection of better nodes, but also explores other nodes that would otherwise be ignored. Additionally, threads are used to divide work between workers and improve solving speeds.

- **Simulated Annealing (SA):** An iterative optimization algorithm inspired by the annealing process in metallurgy. The algorithm begins with a greedy independent set and explores neighboring solutions by adding or removing nodes while maintaining independence. A probabilistic mechanism allows acceptance of suboptimal solutions to escape local optima, with the probability decreasing as the temperature cools. Over multiple iterations, the temperature gradually reduces, refining the solution to converge toward a maximum-weight independent set.

II. Formal Analysis

A. Monte Carlo (MC)

Algorithm 1 Monte Carlo (MC)

Require: Graph $G = (V, E)$ with node weights $w : V \to \mathbb{R}$, Number of iterations I
Ensure: Near-optimal maximum weight independent set S^*

$S^* \leftarrow \emptyset$
max_weight $\leftarrow 0$
isolated_nodes $\leftarrow \{v \in V \mid \deg(v) = 0\}$
$S^* \leftarrow$ isolated_nodes
remaining_nodes $\leftarrow V \setminus$ isolated_nodes
for $i = 1$ to I **do**
    $S \leftarrow$ isolated_nodes
    current_weight $\leftarrow \sum_{v \in \text{isolated\_nodes}} w(v)$
    Randomly shuffle remaining_nodes
    excluded $\leftarrow \emptyset$
    for all $v \in$ remaining_nodes **do**
        if $v \notin$ excluded **then**
            Add v to S
            current_weight $\leftarrow$ current_weight $+ w(v)$
            Add v to excluded
            Add all neighbors of v to excluded
        end if
    end for
    if current_weight $>$ max_weight **then**
        max_weight $\leftarrow$ current_weight
        $S^* \leftarrow S$
    end if
end for
return S^*

This algorithm generates I independent sets S by randomly shuffling the nodes and constructing a valid independent set for each iteration:

- For each iteration:
  - The nodes are randomly shuffled.
  - Process the nodes sequentially:
    - If a node is not in the *excluded* set, add it to the current independent set.
    - Add the neighbors of the selected node to the *excluded* set to maintain independence.
- Calculate the total weight of the independent set.
- If the weight of the current independent set exceeds the maximum weight found so far, update the best solution S^*.

The algorithm terminates after completing T iterations and returns the best solution found, S^*.

During the development of the algorithm, I initially implemented a naive approach to avoid repeated solutions. This method involved hashing each random node order, storing the hashes in a set, and checking each newly shuffled order against the set to determine if it had already been tested. However, the total number of possible random node orders is proportional to $(n - n_i)!$, where n_i represents the number of isolated nodes in the graph G. Given this factorial growth, collisions in these orders are exceedingly rare.

Experimental results further supported this observation: the number of operations required remained virtually unchanged before and after incorporating this mechanism. As a result, the additional computational overhead introduced by hashing and set operations was unwarranted. Consequently, I decided to remove this component from the algorithm, streamlining its implementation without compromising its effectiveness.

Next, we analyze the computational complexity of the algorithm by breaking it down into different layers, starting with the outermost scope. The first *for* loop is executed I times, yielding a complexity of $O(I)$. Still in the outermost scope, a preprocessing step is performed to separate the isolated nodes from the rest of the graph nodes, reducing the amount of operations in the inner loops. This preprocessing requires $O(n)$ operations.

Inside the main loop, the graph nodes are randomly shuffled, which also requires $O(n)$ operations. Additionally, there is another loop that iterates n times, contributing an additional $O(n)$ factor. Therefore, the combined complexity of these nested operations is $O(n + I \cdot (n + n)) = O(n + I \cdot 2n)$, which simplifies to $O(I \cdot n)$.

At the innermost layer, the operations involve basic tasks such as checking for the presence of a node in the excluded set and adding the node to the solution set, both of which are constant-time operations, $O(1)$. The most computationally expensive operation in this layer is adding all the neighbors of the selected node to the excluded set.

The operation of retrieving a node's neighbors and adding them to the excluded set is, at worst, $O(n)$. However, since neighbors are excluded from future iterations and no longer eligible for the solution, this operation is not performed on those nodes. As a result, although the process is technically $O(n)$, , only a subset of $n_s \leq n$ will actually undergo this opera-

tion. This implies that the complexity depends on the average number of neighbors that are still eligible for inclusion in the solution.

Calculating the exact complexity of this part of the algorithm is difficult due to several factors, such as the expected number of non-isolated nodes and the decreasing pool of eligible nodes as the algorithm progresses. Given the constraints on report length and the delivery deadline, I opted to provide an estimation with an oversimplification that still captures the essence of the complexity dynamics.

By simplifying the analysis, assuming that d represents the average degree of a randomly selected node in G and that n_{ni} is the number of non-isolated nodes, we estimate the complexity of this step as $\sim O(\frac{n_{ni}}{d+1})$, where $d = (n-1)k$, with k being the edge probability of the graph.

Thus, the overall computational complexity of the algorithm can be approximated as:

$$O(I \cdot (n \cdot \frac{n_{ni}}{d+1}))$$

This expression indicates that the complexity is higher for the middle values of k, resulting in an upper bound of $O(I \cdot n^2)$, and lower for the extreme values of k, leading to a lower bound of $\Omega(I \cdot n)$.

B. Parallel Heuristic Monte Carlo (PHMC)

Given the similarities between this algorithm and the previous one, the pseudo code will not be shown in full, instead, I will explain the differences between both and demonstrate the changes in pseudo code.

This algorithm differs from the previous one in two significant ways. First, instead of randomly shuffling the nodes, it assigns a probability to each node based on the Weight-to-Degree heuristic (1), explored in the previous report. This means that nodes with higher weight-to-degree ratios have a greater chance of being selected earlier in the process. By doing so, the algorithm prioritizes more valuable nodes while still introducing enough randomness to avoid getting stuck in local optima and maintain a $O(n)$ complexity in shuffling.

Require: Graph $G = (V, E)$ with node weights $w : V \to \mathbb{R}$
Ensure: Shuffled nodes with weighted probabilities
1: **for all** $v \in V$ such that $G.degree(v) > 0$ **do**
2: $ratio \leftarrow \frac{w(v)}{G.degree(v)}$
3: Add $(v, ratio)$ to list *weight_degree_ratio*
4: **end for**
5: Shuffle nodes in *weight_degree_ratio* proportionally to their weights
6: **return** Shuffled nodes

Fig. 1: Weighted Random Node Selection (Simplified)

The second difference comes in the form of parallelism. As explored by the author Michael Luby [4], integrating this concept with Monte Carlo can theoretically improve the algorithm as a whole, and through the correct avenues make it deterministic, however, for the purpose of this work, I maintained the random factor and used it to improve solving times. The way parallelism is implemented in this algorithm is by dividing I_t iterations between t threads, I_t being the number of iterations each thread has to perform.

In regards to the computational complexity, because the foundation of the algorithm is the same as the MC and the weighted shuffle doesn't change the complexity, we can start by deducing that the overall computational complexity is still $O(I \cdot (n \cdot \frac{n_{ni}}{d+1}))$, however, the work is distributed between t threads and solved in parallel, meaning that the **time complexity** is now:

$$O(\frac{I}{t} \cdot (n \cdot \frac{n_{ni}}{d+1}))$$

This approach does not reduce the total number of operations the algorithm must perform, but it significantly accelerates the process of arriving at the optimal solution.

C. Simulated Annealing (SA)

Algorithm 2 Simulated Annealing (SA)

Require: Graph $G = (V, E)$ with node weights, iterations I, initial temperature T_0, cooling rate α
Ensure: Near-optimal maximum weight independent set
1: Initialize *current_set* using a greedy independent set
2: *best_set* ← *current_set*
3: *current_weight* ← Weight(*current_set*)
4: *best_weight* ← *current_weight*
5: $T \leftarrow T_0$
6: **for** $i = 1$ to I **do**
7: **if** Random(0, 1) < 0.5 **then**
8: Add a random valid node to *current_set*
9: **else**
10: Remove a random node from *current_set*
11: **end if**
12: *new_weight* ← Weight(*current_set*)
13: **if** *new_weight* > *current_weight* **or** $\exp((new\_weight - current\_weight)/T) >$ Random(0, 1) **then**
14: Accept the new solution
15: *current_weight* ← *new_weight*
16: **end if**
17: **if** *current_weight* > *best_weight* **then**
18: Update *best_set* and *best_weight*
19: **end if**
20: $T \leftarrow T \times \alpha$ ▷ Cool down temperature
21: **end for**
22: **return** (*best_set*, Operations_Count)

This algorithm is based on the works of Maria Angelini and their team [5], who explore various types of algorithms to find the largest independent set in sparse graphs, one of them being the *Simulated Annealing*. My implementation, however, is not nearly as robust as theirs, as fully replicating their research would require a level of expertise and time that I was not able to allocate to this work.

That said, this algorithm is very different from the others, mainly because each iteration works on refining the same solution rather than generating a new one every time. It begins by using the Monte Carlo algorithm with a single iteration to generate an initial independent set. This set forms the starting point for the optimization process.

The algorithm then iteratively modifies this initial set by either adding or removing nodes while respecting the independence constraint. At each step, a candidate modification is accepted based on its impact on the solution's weight and a probabilistic acceptance criterion. This criterion ensures that, while the algorithm favors improvements, it can also accept worse solutions with a small probability to escape local optima, a core principle of simulated annealing.

Over time, the "temperature" parameter decreases according to a cooling schedule, gradually reducing the likelihood of accepting worse solutions and guiding the algorithm toward convergence.

The computational complexity of this algorithm can be analyzed by considering its key operations:

1. Initialization with Monte Carlo: The algorithm initializes the solution using the Monte Carlo method with 1 iteration. Using the calculated formula, we can deduce that the complexity of this step is:

$$O(n \cdot \frac{n_{ni}}{d+1}),$$

where I is the number of iterations, n is the number of nodes, n_{ni} is the number of non-isolated nodes and d is the average degree of a node.

2. Main Loop: The loop executes I times, performing the following key operations:

- **Neighbor Generation**: This involves either adding or removing a node:
  - Before adding a node, a 'candidates' list is generated to determine which nodes are eligible for addition. This process runs through every node n in the graph, checking if they are not in the solution set ($O(1)$) and if none of their neighbors are in the solution ($O(d)$, d being the average degree of a given node). The complexity of this step is $O(n \cdot d)$.
  - Adding or removing a node from the set is a constant-time operation, $O(1)$.
- **Weight Calculation**: For each candidate solution, the weight of the set is recalculated, which involves summing over the nodes in the solution. If the solution size is s, this operation has complexity $O(s)$.
- **Acceptance Criterion**: The acceptance decision involves evaluating the exponential cooling formula, which has constant complexity, $O(1)$.
- **Temperature Update**: Multiplying the temperature by the cooling rate is a constant-time operation, $O(1)$.

Combining these, each iteration has a worst-case complexity of approximately:

$$O(n \cdot d),$$

3. Total Complexity: The total complexity is the sum of the initialization cost and the main loop cost:

$$O(n \cdot \frac{n_{ni}}{d+1}) + O(I \cdot n \cdot d) \approx O(I \cdot n \cdot d)$$

For denser graphs, the complexity of this algorithm will tend to $O(n^2)$, and for sparser graphs it will tend to $O(n)$.

III. Experiments

All algorithms and tests were developed in Python. However, due to the parallelism requirements of the PHMC algorithm, a newer and experimental version of Python, 3.13.0t, was utilized. This version introduces free threading, enabling each thread to access its own Python interpreter and thus allowing true parallelism.

The performance improvement with Python 3.13.0t was significant compared to Python 3.12.5, where parallelism remained constrained by the Global Interpreter Lock (GIL). Prior to upgrading, I also attempted to implement the algorithm using the multiprocessing module in Python 3.12.5. However, the overhead of spawning new processes and transferring data between them severely impacted execution time, making the approach inefficient for this use case.

The adoption of Python 3.13.0t not only resolved these bottlenecks but also demonstrated the potential of free threading to significantly enhance performance in parallel algorithms. It also greatly affected the timeline of the work, because some modules didn't yet have a version that was compatible with this advancement, and dealing with these problems took a lot of time.

Another significant change compared to the previous report was the method of storing graphs. Instead of using Pickle, graphs were stored in JSONL format, allowing them to be loaded individually during the solving process. This approach enhanced both performance and memory efficiency..

All randomized algorithms were configured to run for 2,500 iterations, a value determined through testing to

provide a balance between accuracy and computational efficiency. For the Simulated Annealing algorithm, the initial temperature was also set to 2,500, as testing showed the algorithm performed best when the initial temperature matched the number of iterations.

For the PHMC algorithm, parallelism was implemented using up to 8 threads, the most adequate amount for my machine. For smaller graphs with fewer than 8 nodes, fewer threads were utilized, ensuring optimal resource usage on the available hardware.

A. Monte Carlo Complexity

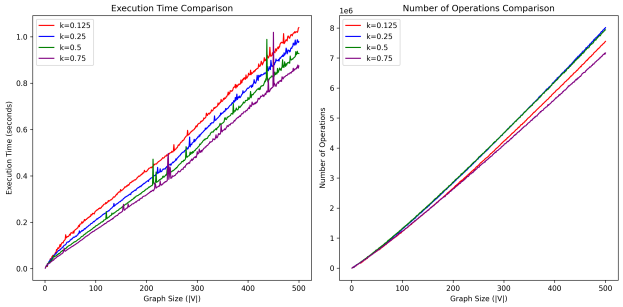

Fig. 2: Monte Carlo Execution Time And Operations Analysis For Different k Values

In this figure, we observe two key trends: the execution time scales approximately linearly with graph size, while the number of operations follows part of the theoretically calculated complexity, which is evident by $k = 0.125$ having less operations than $k = 0.25$ and $k = 0.50$.

The discrepancy between execution time and operation count can be attributed to several factors:

- **Operation Complexity**: Not all operations have equal computational cost. Certain operations, such as verifying independence or managing data structures, may require more processing time than others.
- **Implementation Overheads**: The overhead from Python's runtime environment, memory allocation, or garbage collection can introduce variability in execution time that is independent of the raw operation count.
- **Randomized Nature of Monte Carlo**: The Monte Carlo algorithm's behavior can vary based on randomness, causing execution time to fluctuate even if the number of operations remains consistent, although, in this case, this seems unlikely to be the cause.

It is also relevant to mention that all values of k seem to lead to a linear progression, which was not expected to be the case, according to the formal analysis. This may be due to the oversimplification that was done in that step, which may suppress why the algorithm has this behavior.

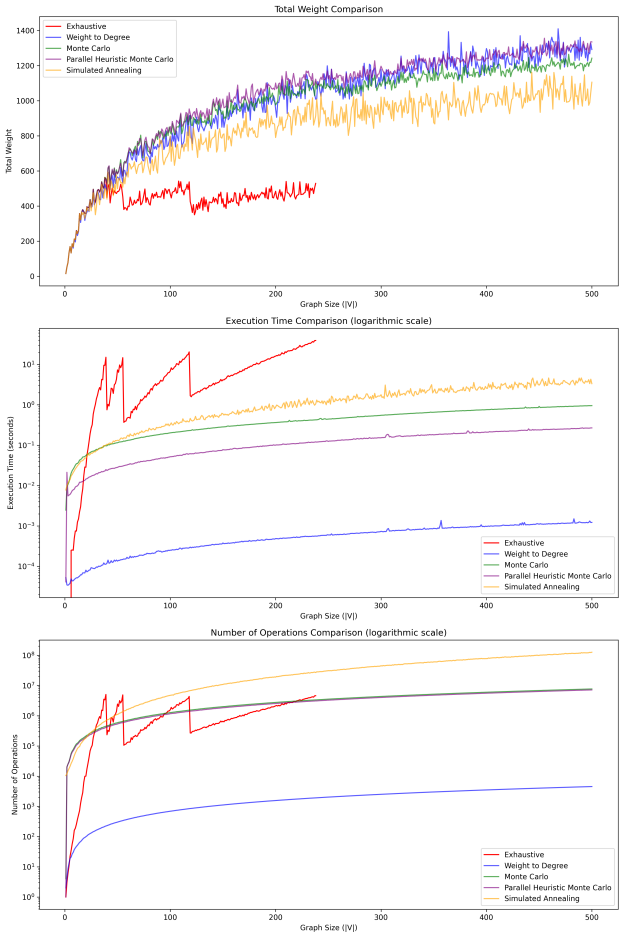

Fig. 3: Algorithm Comparison (Average) for k in [0.125, 0.25, 0.5, 0.75] - Total Weight, Execution Time (log scale), Number of Operations (log scale)

B. Algorithm Comparison

Disclaimer: *The graphs representing the 'Execution Time' and the 'Number of Operations' are in logarithmic scale. Also, because the graphs show the average for all values of k, the exhaustive algorithm suffers from misrepresentation in the 'Total Weight' metric. The drops that happen in the two other metrics correspond to the exhaustive algorithm reaching its limit for a certain k value.*

For comparison purposes, the exhaustive algorithm and the most effective greedy heuristic were selected as benchmarks for the developed randomized algorithms. The graphs above compare the algorithms based on solution quality, execution time, and computational efficiency, from which we can conclude the following:

Solution Quality

- Exhaustive Search delivers suboptimal results for large graph sizes due to its inability to scale, often timing out as graph size increases.
- Weight to Degree, a simple greedy heuristic, achieves consistent results and is able to compete with the randomized algorithms.

- Monte Carlo initially provides better results than Weight to Degree but fails to maintain that difference on larger graphs.
- Parallel Heuristic Monte Carlo consistently achieves the best or near-best solutions, leveraging parallelism for improved performance.
- Simulated Annealing produces the worst results out of all the non-exhaustive algorithms, and continues to decline for larger graphs.

Execution Time and Efficiency

- Exhaustive Search is computationally prohibitive, with execution time and operations growing exponentially.
- Weight to Degree is the fastest, maintaining linear-like scalability, making it ideal when speed is critical.
- Monte Carlo and Parallel Heuristic Monte Carlo balance solution quality with efficiency, with the latter benefiting from parallelism to scale effectively.
- Simulated Annealing requires the most computational effort among the heuristics, with slower execution times as graph size increases.

Overall, for large and complex graphs, Parallel Heuristic Monte Carlo offers the solutions and maintains a good scalability, while Weight to Degree is preferable for fast and near-optimal solutions when runtime is critical. The Simulated Annealing algorithm fails in every front, providing worse solutions in slower times.

C. Accuracy

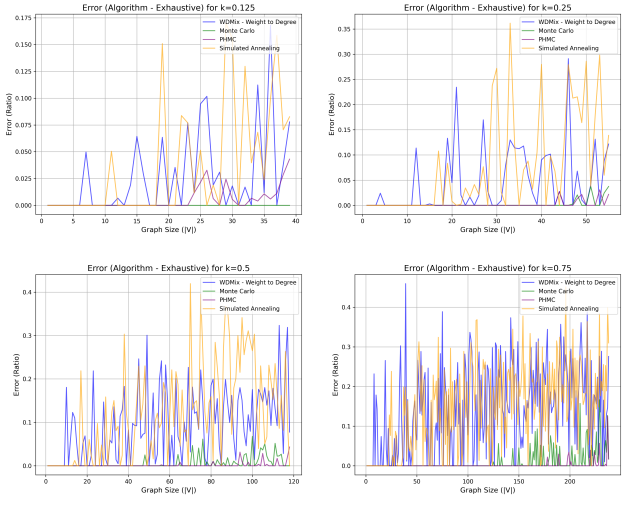

Fig. 4: Error Ratio Of Algorithms For Each Value Of k

	k=0.125			k=0.25		
Algorithm	Average Deviation	Maximum Deviation	Accuracy	Average Deviation	Maximum Deviation	Accuracy
Weight to Degree	17.77	142	0.49	26.58	200	0.44
Monte Carlo	0.00	0	1.00	1.53	26	0.91
PHMC	4.05	36	0.67	1.44	24	0.91
Simulated Annealing	26.28	128	0.56	47.82	232	0.42

TABLE I: Deviation and Accuracy Metrics for k=0.125 and k=0.25

	k=0.50			k=0.75		
Algorithm	Average Deviation	Maximum Deviation	Accuracy	Average Deviation	Maximum Deviation	Accuracy
Weight to Degree	46.72	197	0.24	58.88	207	0.18
Monte Carlo	3.49	41	0.75	4.49	88	0.82
PHMC	0.95	29	0.90	0.83	63	0.95
Simulated Annealing	63.18	224	0.25	61.90	232	0.19

TABLE II: Deviation and Accuracy Metrics for k=0.50 and k=0.75

Fig. 4 illustrates the error ratio when comparing each algorithm's solution weight to the optimal solution obtained by the exhaustive algorithm. A higher ratio indicates a greater deviation from the optimal solution, suggesting a less accurate heuristic.

TABLE I and *TABLE II* provide further insights into the deviations and accuracy of the algorithms. The first column displays the average deviation, identifying which algorithm generally comes closest to the optimal solution. The second column presents the maximum deviation, highlighting the algorithm with the largest outliers. Finally, the third column shows accuracy, defined as the percentage of instances in which each algorithm finds the optimal solution, relative to the total number of graphs solved.

From this data, we can better understand which algorithm performs better:

- The Weight-to-Degree and Simulated Annealing algorithms generally exhibit lower accuracy, with Simulated Annealing showing larger deviations than Weight-to-Degree.
- As anticipated, the Monte Carlo and PHMC algorithms outperform the others by a substantial margin, achieving very low deviations and near-perfect accuracy. Notably, Monte Carlo achieved 100% accuracy for $k = 0.125$ in one instance.
- PHMC emerges as the strongest performer overall, except for $k = 0.125$, where its metrics drop significantly. This unexpected result may indicate that the algorithm encountered unusually poor outcomes on these specific graphs.
- Across all algorithms, error rates tend to increase as k grows. However, the Monte Carlo-based algorithms demonstrate remarkable resilience, maintaining high accuracy and often achieving optimal solutions despite the increasing challenge.

D. Largest Graph

Due to compatibility issues between the most recent version of the package NetworkX and the Python version 3.13.0t, graph creation and loading was unfeasible, taking almost 10-15 times more than when using stable versions of Python. Because of that, and because I couldn't find a suitable pre-generated graph in time, I wasn't able to use free threads, making the PHMC algorithm seem less efficient than it is.

As seen in *Fig. 5*, the naive greedy method, using the best heuristic determined by the previous report, is able to solve the graph with ease in less than 0.03 seconds. The randomized algorithms, Monte Carlo and PHMC, take significantly longer, by a factor of

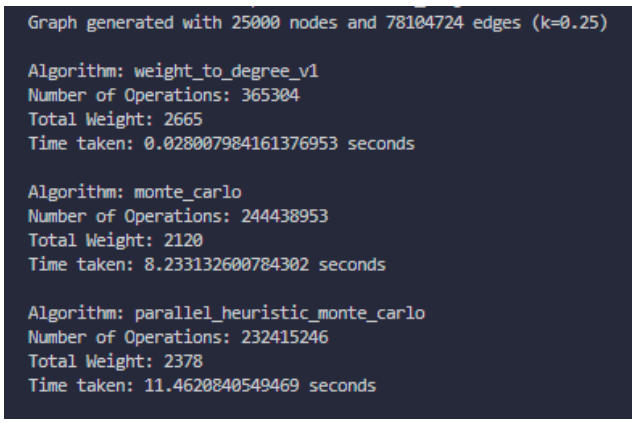

Fig. 5: Largest Feasible Graph: n = 25000, k = 0.25

approximately 312. Still, it's a remarkable speed given the advantage they have on the greedy algorithm in terms of accuracy. The Simulated Annealing algorithm is not feasible for this graph size.

In terms of solution quality, the greedy method manages to achieve a higher total weight than the randomized algorithms, which is very surprising. This may suggest that the rate at which these algorithms lose accuracy is slow at the start but eventually become worst than the heuristic.

Through testing, I also determined that the PHMC is usually 3.32 to 3.91 times faster than the Monte Carlo algorithm, depending on k, achieving faster times for lower values of k.

IV. Conclusion

In this study, I focused on evaluating randomized algorithms for solving the Maximum Weighted Independent Set (MWIS) problem, using the exhaustive algorithm and the WDMix heuristic as benchmarks.

Monte Carlo emerged as a strong contender, delivering near-optimal solutions with minimal deviation from the exhaustive algorithm, particularly for smaller graph sizes. Its accuracy and simplicity make it a reliable choice for a variety of graph types. Parallel Heuristic Monte Carlo (PHMC) expanded on this by utilizing parallelism, significantly improving execution times while maintaining comparable accuracy. However, when solving the largest graph, its effectiveness came into question, warranting further investigation.

Simulated Annealing, on the other hand, showed potential in exploring diverse solution spaces, often finding high-quality solutions. However, its performance was less consistent, with greater deviations from the optimal solutions, particularly in denser graphs, where it sometimes struggled to maintain accuracy.

In conclusion, while the exhaustive algorithm and WDMix heuristic remain invaluable as benchmarks and for understanding the problem space, the randomized algorithms explored in this study offer practical and scalable solutions for MWIS. Monte Carlo and its threaded variant strike an excellent balance between accuracy and efficiency, making them the preferred choices for larger graph instances. Simulated Annealing, despite its variability, provides a robust alternative in scenarios where solution diversity and exploratory potential are prioritized.

V. Notes and Future Work

Here are some notes on the topics of this report, and some suggestions on what I would do if I continued this work:

- **Simulated Annealing**: This algorithm could use more work, especially when it comes to selecting a new node to add. I tried to implement a set where I kept the conflicted nodes, in order to diminish the computational complexity, but I didn't manage to get it working correctly.
- **Python 3.13.0t**: As evident throughout the work, this version of Python is still in early development and is incompatible with some of the packages I used. This comes as a consequence of developing a truly parallel algorithm in Python, and if I had known I would've opted by developing an alternative algorithm.
- **Monte Carlo Based Algorithms**: The *Largest Graph* section demonstrated that these algorithms can still be surpassed by greedy methods. Further research is needed to understand this phenomenon. Factors such as the randomized algorithms' parameter tuning, their convergence rates, and the inherent structure of larger graphs likely contribute to this surprising outcome. By addressing these aspects, future work can determine whether these algorithms can be adapted or improved to retain their advantage over heuristic methods, even at larger scales.
- **Graph Generation**: Three datasets of graphs were utilized in this study: two were generated, and one was pre-existing [6]. The generated datasets consisted of small graphs (1–500 nodes) and large graphs (100–3500 nodes). The pre-generated dataset included over 500 graphs intended for algorithm comparison. Ultimately, only the small dataset was actively used in the experiments, though results for the other datasets are included in the data.

References

[1] Deepak Puthal, Surya Nepal, Cecile Paris, Rajiv Ranjan, and Jinjun Chen, "Efficient algorithms for social network coverage and reach", *Proceedings - 2015 IEEE International Congress on Big Data, BigData Congress 2015*, pp. 467–474, 8 2015.

[2] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe, "Efficient traversal of mesh edges using adjacency

primitives", *ACM Transactions on Graphics*, vol. 27, 12 2008.
URL: https://dl.acm.org/doi/10.1145/1409060.1409097

[3] Richard M. Karp, "An introduction to randomized algorithms", *Discrete Applied Mathematics*, vol. 34, pp. 165–201, 11 1991.

[4] Michael Luby, "Simple parallel algorithm for the maximal independent set problem.", *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 1–10, 1985.
URL: https://dl.acm.org/doi/10.1145/22145.22146

[5] Maria Chiara Angelini and Federico Ricci-Tersenghi, "Monte carlo algorithms are very effective in finding the largest independent set in sparse random graphs", *Physical Review E*, vol. 100, 4 2019.
URL: https://ui.adsabs.harvard.edu/abs/2019arXiv190402231C/abstract

[6] Ernesto Parra Inza, "Random graph", 2024.
URL: https://doi.org/10.17632/rr5bkj6dw5.7