Use of a Randomized Algorithm on the minimum cut graph problem

Pedro Ponte
Licenciatura em Engenharia Informática

***Abstract* –Find a minimum cut for a given undirected graph G(V, E), with n vertices and m edges. A minimum cut of G is a partition of the graph's vertices into two complementary sets S and T, such that the number of edges between the set S and the set T is as small as possible.**

***Keywords* –Graph, Minimum Cut, Computational Complextity, Randomized Algorithms**

I. Introduction

A. Minimum Cut

The minimum cut problem consists in taking an undirected, unweighted graph and finding the smallest set of edges whose removal splits the graph in two complimentary sets

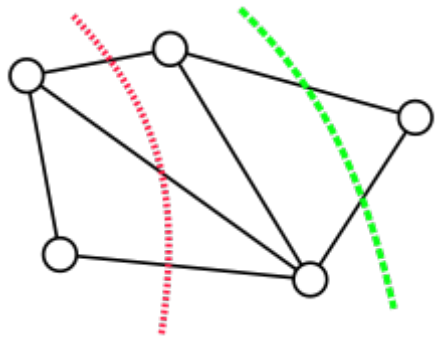

Fig. 1: Example with a cut of three edges represented by a dotted red line and a minimum cut represented by a dotted green line.

In this example if we followed the **red line** we would end with **a cut weight** of 3, since we cut 3 edges. However, if we choose the **green line** our cut weight is 2, and therefore we have our minimum cut.

B. Randomized Algorithms

A *Randomized Algorithm* [1] [2] [3] is an algorithm that uses randomness as part of its logic, making decisions by using random bits as auxiliary input (like tossing coins). Unlike deterministic algorithms, their behavior can vary between different executions on the same input.
These algorithms are

- More efficient than deterministic alternatives in many cases
- Simpler to implement for certain complex problems
- Capable of providing good average-case performance
- Subject to a controlled probability of error

which means that while they might not guarantee the optimal solution every time, they can find very good solutions quickly with high probability. A classic example is comparing two databases: while a deterministic approach requires transferring the entire database (n bits), a randomized approach needs only $O(\log n)$ bits with a negligible probability of error.

II. Algorithm

A. Karger's Algorithm

The *Karger's Algorithm* [4] is a randomized algorithm to compute a minimum cut of a connected graph through random edge contraction. Instead of methodically searching for a cut, it repeatedly contracts random edges until only two vertices remain, forming a cut.

The algorithm works by:

- Selecting edges uniformly at random
- Contracting the vertices connected by the selected edge
- Maintaining vertex groups to keep track of which original vertices are in each contracted vertex
- Continuing until only two vertices remain, whose edges give us the cut

When contracting an edge (u,v):

- The vertices u and v are merged into a single vertex
- Any edges between u and v are removed
- All other edges incident to u or v become incident to the new merged vertex

Since the algorithm makes random choices, it must be run multiple times to increase the probability of finding the minimum cut. Each run may produce a different cut, but the smallest cut found is saved

III. Analysis

A. Formal Analysis

To better understand how our algorithm will scale we'll do a *Formal Analysis* before analysing the results, this will give an estimate of how long it will take to run and also give us a way of rating it in terms of performance.

A.1 Time complexity

1. for each iteration - O(n^2)
   - contract edges until two vertices remain - O(n)
   - for each contraction
     - select random edge - O(1)
     - merge vertices and update edges - O(n)
   - calculate cut size - O(n^2)
2. perform n^2 log n iterations for high probability

This gives us a total complexity of:

$$O(n^2 \log n \cdot n^2) = O(n^4 \log n) =$$

$$\sum_{i=1}^{n^2 \log n} \left[\sum_{j=1}^{n} \left(1 + n + \sum_{k \in \text{set}_a} \sum_{l \in \text{set}_b} 1 \right) \right]$$

B. Probability Analysis

Let G = (V,E) be our graph with minimum cut size k.

B.1 Single Iteration Probability

For a specific minimum cut C, the probability of success in one iteration is:

$$P(\text{success}) \geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i} \right)$$

This expands to:

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \binom{n}{2}^{-1}$$

B.2 Multiple Iterations

Running the algorithm T = $\binom{n}{2} \ln n$ times, the failure probability becomes:

$$P(\text{failure}) = \left[1 - \binom{n}{2}^{-1} \right]^{\binom{n}{2} \ln n} \leq \frac{1}{n}$$

B.3 Comparison to Random Selection

This is significantly better than random cut selection, which has probability:

$$P(\text{random success}) \leq \frac{\binom{n}{2}}{2^{n-1} - 1}$$

B.4 Total Runtime

For a graph with m edges, the total runtime across all iterations is:

$$O(T \cdot m) = O(n^2 m \log n)$$

C. Experimental analysis

For this analysis, the graphs were created using the package **NetworkX** [5], specifically based on the method *erdos_renyi_graph*, using my NMec **98059** as a seed. Code for graph generation can be found in **utils.py**, while the algorithm code, as well as the code to create the number of operations and execution time plots, can be found in **mincut.py**. For the plot generation, **Matplotlib** [6] was used.

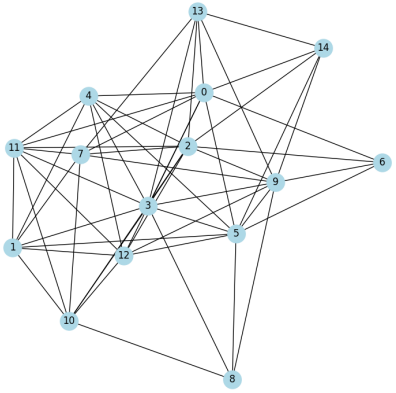

Fig. 2: Generated graph with 13 nodes

C.1 Operation count

To confirm our formal analysis, we can count the number of operations our algorithm goes through depending on number of nodes

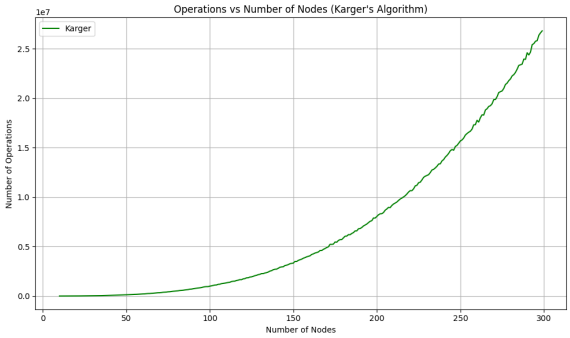

Fig. 3: Number of operations based on graph nodes with regular scale

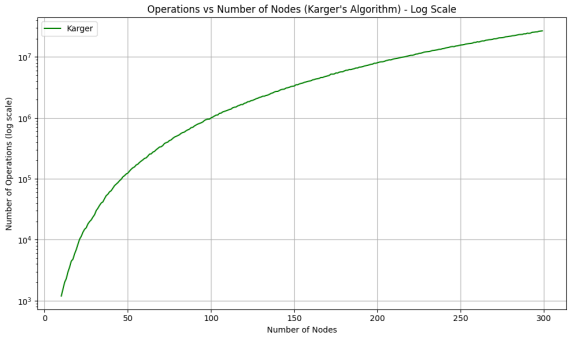

Fig. 4: Number of operations based on graph nodes with logarithmic scale

The complexity is confirmed by looking at the logarithmic scale plots, where we can see an approximately curved line which represents polynomial growth.

C.2 Time measurement

For measuring time we'll check the time it takes for an increasing number of nodes

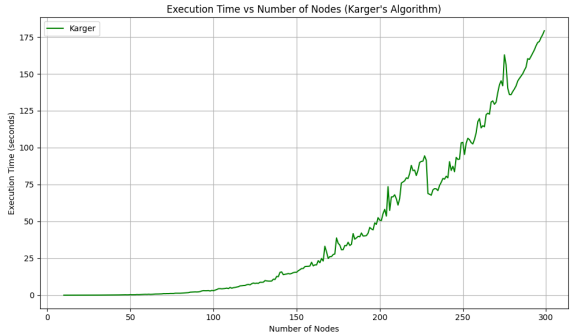

Fig. 5: Execution time based on graph nodes with regular scale

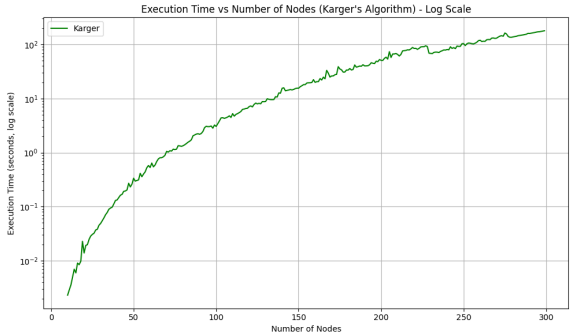

Fig. 6: Execution time based on graph nodes with logarithmic scale

Once again we can confirm that time grows as expected in the formal analysis. As the number of nodes increases, so does the execution time.

Number of Nodes	Time (seconds)
20	0.013
40	0.114
60	0.426
80	1.130
100	2.721
120	5.173
140	10.490
160	17.773
180	32.053
200	46.828
212	52.968

TABLE I: Execution time of Karger's algorithm for different graph sizes

Node Range	Time Increase (s)	Factor Increase
20 → 40	0.101	8.8x
40 → 60	0.312	3.7x
60 → 80	0.704	2.7x
80 → 100	1.591	2.4x
100 → 120	2.452	1.9x
120 → 140	5.317	2.0x
140 → 160	7.283	1.7x
160 → 180	14.280	1.8x
180 → 200	14.775	1.5x

TABLE II: Time increase analysis between node intervals

C.3 Maximum size

These calculations were run on my machine: **System Specifications:**

- **CPU:** Intel(R) Core(TM) i5-11400H @ 2.70GHz (12 cores)
- **RAM:** 15GB Total (7.6GB Available)
- **OS:** Ubuntu 22.04.5 LTS
- **Kernel:** 6.8.0-48-generic
- **GPU:** NVIDIA GeForce RTX 3060 Mobile / Max-Q, Intel UHD Graphics

and the biggest graph it could handle with a reasonable time limit was:

- **300 nodes** - 178 seconds

C.4 Edge probability

Another interesting observation we can make is that when increasing the edge creation probability of each vertex, making the graph more dense, the minimum cut weight increases. Although, by comparing how the cut sizes grow for both our algorithm for the the **Stoer Wagner** [7] algorithm, we can see that they're growth is very similar, so the graph being more dense does not necessarily mean the algorithm will be less efficient.

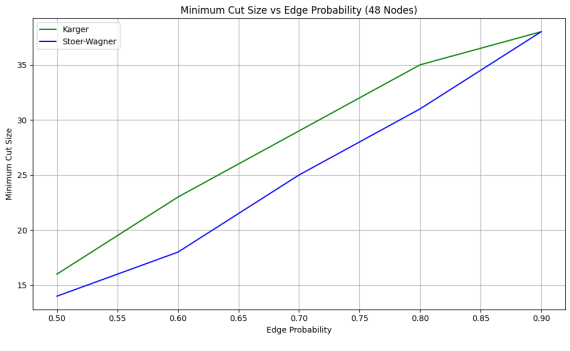

Fig. 7: Minimum cut size based on edge probability of a graph of 48 nodes

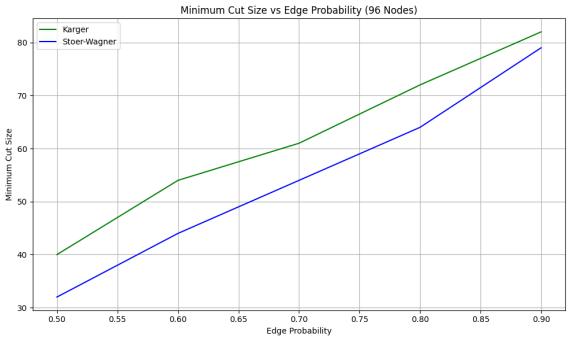

Fig. 8: Minimum cut size based on edge probability of a graph of 96 nodes

C.5 Solution Accuracy

To test the accuracy of our solution we can compare our algorithm to the **Stoer Wagner** algorithm, since it always returns smallest cut possible. For these tests we established a maximum of 100 nodes, with each number of nodes being tested 30 times.

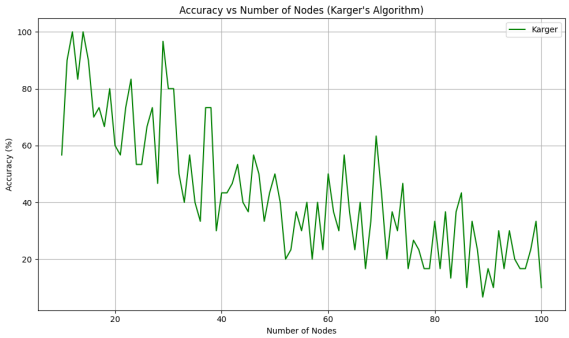

Fig. 9: Accuracy testing with 100 nodes for an average of 30 runs for each node

C.6 Time estimation

To check how much time it would take to solve huge instances of this problem a fit was made using the above calculated values and these were the resulting times. Everything used to calculate these times can be checked by running **time_estimation.py** which uses **scipy optimize** [8]. Although the times are not exact, they give us a good insight on how the algorithm would behave for very large problems.

TABLE III: Karger's Algorithm Predicted Execution Times

Nodes	**Estimated Exec Time**
150	14.64 seconds
200	43.71 seconds
250	101.40 seconds (1.69 minutes)
300	200.40 seconds (3.34 minutes)
400	588.00 seconds (9.80 minutes)
500	1354.20 seconds (22.57 minutes)
750	6156.00 seconds (1.71 hours)
1000	18072.00 seconds (5.02 hours)

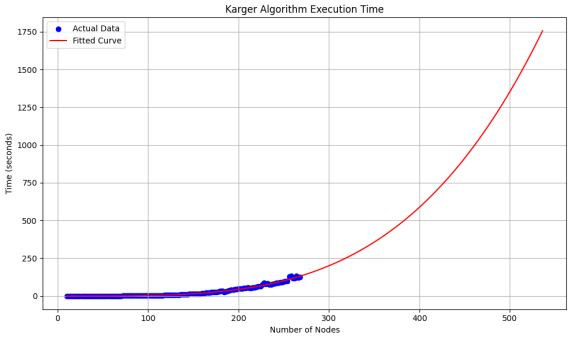

Fig. 10: Fit found based on actual data

IV. Extra results

A. Minimum cut solutions

Some computations of the algorithm for 3 different sizes.

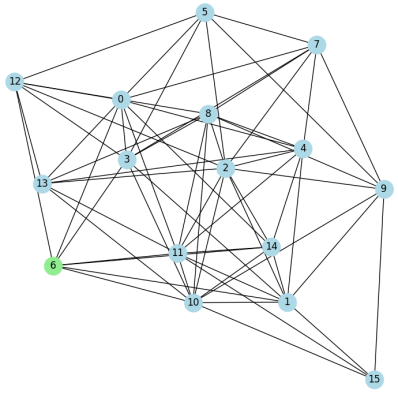

Fig. 11: Solution for graph with 16 nodes

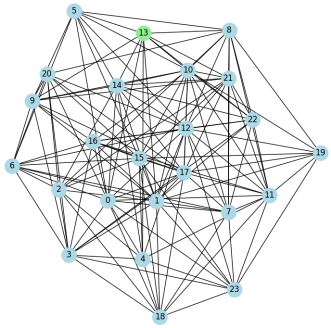

Fig. 12: Solution for graph with 24 nodes

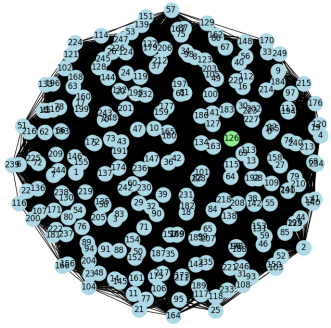

Fig. 13: Solution for graph with 250 nodes

B. SW Graphs

Additionally, a few tests were run for larger graphs provided by the teacher (file **SW_ALGUNS_GRAFOS**), these were the results:

- **SWtinyG** - 13 vertices, 13 edges
  - This graph is too small and it is not connected, so our solution is not really relevant

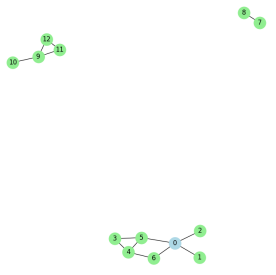

Fig. 14: Solution for the tiny graph

- **SWmediumG** - 250 vertices, 1273 edges
  - Minimum cut: 2
  - Time taken: 10.9 seconds
- **SWlargeG** - 1000000 vertices, 7586063 edges
  - This was not solvable in 2 hours. Considering the estimates calculated before, it would take a very long time before the solution could be found, unless we got really lucky.

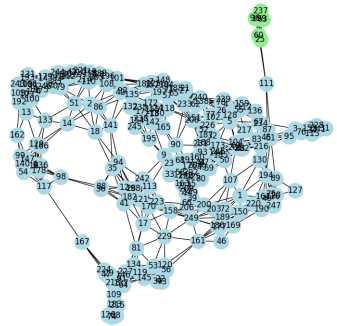

Fig. 15: Solution for the medium graph

V. Conclusion

The formal analysis was proven with the execution time plots, which shows a good analysis of the problem at hand, allowing us to estimate the performance of the algorithm. Our experimental analysis shows the difference between a **randomized** approach and the **exhaustive** used in the first project, with the randomized one being a lot faster, at the cost of not finding the best result. For our accuracy finding, the randomized approach does not seem a better strategy than the **greedy heuristic search** studied before, but **greedy** results could have been misleading as stated in the previous report, as it probably used python cache to speed up the process.

References

[1] Joaquim Madeira, "Introduction to Randomized Algorithms I", Course notes, University of Aveiro, Oct. 2024.

[2] Joaquim Madeira, "Introduction to Randomized Algorithms II", Course notes, University of Aveiro, Oct. 2024.

[3] Joaquim Madeira, "Introduction to Randomized Algorithms III", Course notes, University of Aveiro, Nov. 2024.

[4] "Karger's algorithm - Wikipedia — en.wikipedia.org", `https://en.wikipedia.org/wiki/Karger%27s_algorithm`, [Accessed 30-11-2024].

[5] "NetworkX 3.4.2 documentation — networkx.org", `https://networkx.org/documentation/stable/reference/index.html`, [Accessed 30-11-2024].

[6] "Matplotlib documentation; Matplotlib 3.9.3 documentation — matplotlib.org", `https://matplotlib.org/stable/index.html`, [Accessed 02-12-2024].

[7] "Stoer–Wagner algorithm - Wikipedia — en.wikipedia.org", `https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner_algorithm`, [Accessed 01-12-2024].

[8] "Optimization and root finding (scipy.optimize); SciPy v1.14.1 Manual — docs.scipy.org", `https://docs.scipy.org/doc/scipy/reference/optimize.html`, [Accessed 03-12-2024].