+2st Project: Minimum Weight Cut with Randomized Algorithms

Advanced Algorithms

José Miguel Costa Gameiro, 108840

Resumo **- This document presents a brief study conducted within the Advanced Algorithms course to analyse Randomized Algorithms when used to solve the Minimum Weight Cut Graph Problem. A computational complexity analysis is performed using the execution time and the number of operations taken. The formal analysis is also compared with the experimental.**

I. INTRODUCTION

This report was written as a result of the **second project** proposed in the Advanced Algorithms course and it aims to present a brief description of the Minimum Weight Cut Graph Problem, what is a random algorithm and how to use these types of algorithms to solve this problem. It is also presented some key concepts about Graphs and some other important aspects

II. CONTEXTUALIZATION

A. Graphs

A graph is a mathematical structure used to model relationships or connections between objects. It consists of:

- **Nodes (or Vertices):** The object entities in the graph;
- **Edges:** The connections or relationships between pairs of vertices.

Graphs can be classified based on their properties:

- **Direct Graphs (Digraphs):** The edges have a direction, indicating the relationship flows from one node to another;
- **Undirected Graphs:** The edges do not have a direction, meaning the relationship is bidirectional;
- **Weighted Graphs:** Each edge has a weight or cost, representing the strength or importance of the connection;
- **Unweighted Graphs:** All edges are considered equal, with no weights.

Graphs are widely used in computer science, mathematics, and other fields to represent networks, transportation systems, or even the relationships in molecules.

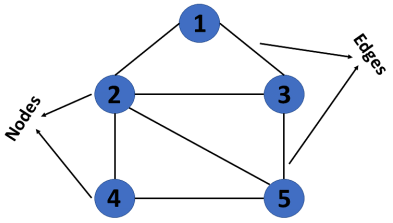

Fig. 1 - Example of a simple graph

B. Graph Representations

A graph can be represented in various ways, but two primary methods are commonly used:

1. **Adjacency List:** uses an array where each element contains the address of a linked list. The first node in the linked list represents a node, and the subsequent nodes in the list represent the vertices it is connected to. This approach is memory-efficient and works well for sparse graphs;
2. **Adjacency Matrix:** uses a 2D array of size $V * V$, where V represents the number of nodes in the graph. The value at a given position $adj[i][j] = 1$ indicates the presence of an edge between node i and node j. For undirected graphs, the adjacency matrix is symmetric. Additionally, adjacency matrices can be used to represent weighted graphs, where $adj[i][j] = w$ and w is the weight of the dge between i and j. This representation is useful for dense graphs but can be memory-intensive for sparse graphs.

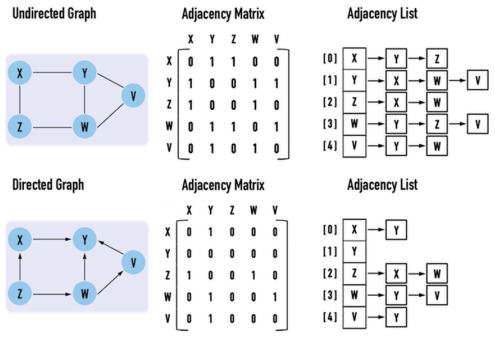

Fig. 2 - Adjacency List vs. Adjacency Matrix

C. Connected Graphs

A **connected graph** is a type of graph in which there is a path between every pair of vertices. In other words, it is possible to traverse the graph and reach any vertex starting from any other vertex.

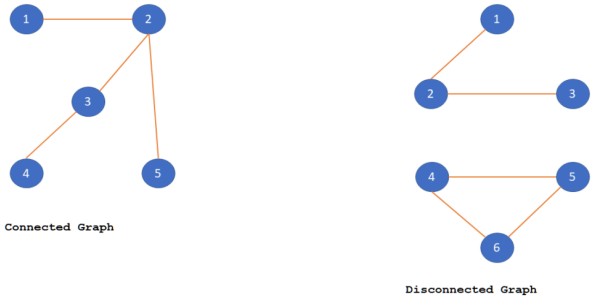

Fig. 3- Connected vs Disconnected graphs

III. PROBLEM CONTEXT

A. Minimum Weight Cut Problem

The **minimum weight cut problem** in graph theory involves finding the smallest total weight of edges that need to be removed to divide a graph into two disjoint subsets, effectively disconnecting it. Given a graph where each edge has an associated weight, the goal is to identify a partition of the vertices such that the sum of the weights of edges crossing between the two subsets is minimized.

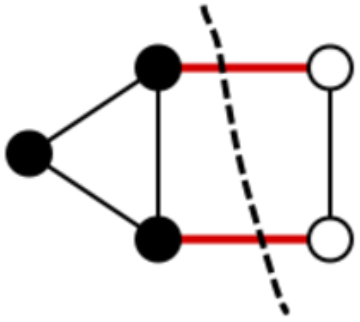

Fig. 4- Minimum weight cut solution example

B. Randomized Algorithms

To solve the min weight cut problem, two algorithms were used (one of them developed, the other used from a **Python**'s library) the **Stoer-Wagner** and **Karger's algorithms**, which are based on a randomized approach.

A **randomized algorithm** is an algorithm that employs random numbers to influence its decision-making process during execution. These algorithms are **non-deterministic**, meaning they can produce different outputs or follow different execution paths for the input on different runs. Randomized algorithms are widely used because they often simplify complex problems, provide faster solutions on average, and are particularly effective in scenarios involving large data sets or high levels or uncertainty.

The **Stoer-Wagner algorithm** is a deterministic algorithm for solving the problem presented for an undirected and weighted graph. It iteratively constructs cuts between two sets of nodes and updates the graph to progressively refine these cuts. At each iteration, it finds the "most strongly connected" pair of vertices and merges them, gradually identifying the minimum cut. Since this algorithm is a deterministic algorithm meaning that it always guarantees the optimal solution, it was not used in this study.

IV. KARGER'S ALGORITHM

Karger's algorithm is also a randomized algorithm for finding the minimum weight weight cut in an undirected graph. It repeatedly contracts randomly selected edges in the graph until only two nodes remain, at which point the edges between these two nodes define a cut. The algorithm achieves a high probability of finding the minimum cut by running multiple interactions and selecting the smallest cut obtained across all iterations. This algorithm presents two main concepts:

- **Supernode**: group of nodes;
- **Superedge**: all edges between a pair of nodes.

In applying this algorithm to the current problem, the process begins with each node serving as its own **supernode**, and each edge **superedge** consisting of a single edge. The algorithm's primary objective is to randomly select and from all available edges, merge the connected nodes into a single supernode, and repeat the process until two supernodes remain. These final two nodes define the cut.

It is important to refer that this algorithm does not always return the correct solution, however it is faster comparing with brute force, that always found the correct solution

When two nodes are merged the connections between them are kept, except when the merge produces redundant edges. In that case, they are removed. Also, if there are multiple edges between the merged nodes, they are also removed.

```
function Karger-MinCut(graph, num_trials):
    while number of nodes in the graph > 2:
        edge = random choice (edges)
        graph.contract(edge)
    cut = 0
    for i in edges of graph:
        cut = cut + i.weight
    return cut
```

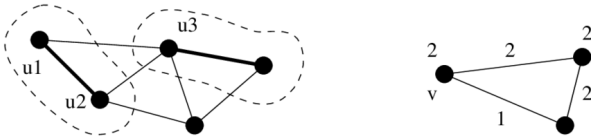

Fig. 5- Example of a contraction of nodes

A. Probability of Failure and Success

In this context, the probabilities of success and failure are defined by the algorithm's correctness when finding the solution. If the algorithm executes 100 times and in 99 of them it provides the correct solution, the algorithm has a 99% rate of success and a 1% of failure.

In this case the probability of returning a minimum cut is defined by the following formula $Psuc \geq \frac{1}{(n\,2)}$ and the probability of failure is $Pfail \leq 1 - \frac{1}{(n\,2)}$.

It is correct to assume that:

- $1 - x = e^{-x}$
- $(n\,2) = \frac{n(n-1)}{2} \leq n^2$

Instead of executing the algorithm only one time, if it is executed **k** times, the probability of failure is

$$Pfail \leq (1 - \frac{1}{(n\,2)})^k$$

where **n** represents the number of nodes.

Using the formulas mentioned above, it is possible to conclude that:

$$Pfail \leq (1 - \frac{1}{n^2})^k \equiv (e^{-\frac{1}{n^2}})^k$$

If the algorithm is executed n^2 times, then the following formula can be obtained:

$$Pfail \leq (e^{-\frac{1}{n^2}})^{n^2} \leq (e^{-\frac{n^2}{n^2}}) \leq e^{-1} \leq \frac{1}{e}$$

So the probability of failure can be given by:

$$Pfail \leq \frac{1}{e}$$

It is possible to go even further, executing the algorithm $n^2 log(n)$.

$$Pfail \leq (e^{-\frac{1}{n^2}})^{n^2 log(n)} \leq (e^{-\frac{n^2 log(n)}{n^2}}) \leq e^{-log(n)} \leq n^{-1}$$

In this case the probability of failure is:

$$Pfail \leq \frac{1}{n}$$

where **n** represents the number of nodes in the graph.

After this explanation, it is possible to conclude that, on one hand, the more this algorithm is executed the lower is the **Probability of failure**. On the other hand,

$$Pfailure = 1 - Psuccess$$

So the probability of success is going to be higher

V. COMPLEXITY ANALYSIS

Usually, random algorithms take less execution time, but not always provide the optimal solution. Karger's algorithm experiments run time of $\theta(n^2)$, since each merge operation takes $\theta(n)$ time (iterating through at most $\theta(n)$ edges and nodes), and there are **n - 2** merges until there are **2 supernodes left**.

A. Graphs Used

To solve the minimum weight cut problem, the graphs were randomly generated as it was made in the first assignment. It was generated graphs with nodes between **4** to **456**. For each of these graphs the number of edges were **12,5%, 25%, 50%** and **75%** of the number of nodes. In the end, more than 1850 were used for testing purposes. These graphs have the same positions and edge weights as the ones used in the first assignment.

The course teacher also provided three files with graphs that can be used in this problem: **SWtinyG.txt, SWmediumG.txt** and **SWlargeG.txt**. After these graphs were built and drawed, it is possible to conclude that the tiny file doesn't provide a connected graph, so in the context of this problem it cannot be used.

It is also not possible to use the large graph since it has more than **7.5 million edges**, **1 million nodes** and this algorithm is executed until it reaches just two nodes. Thus the medium graph is processed and the information is recorded in a file.

B. Number of Basic Operations

The number of basic operations in Karger's Algorithm directly depends on the number of nodes in the graph. Each time an edge is selected and contracted, the total number of nodes decreases by one, with each contraction representing a basic operation. The algorithm continues

until only two nodes remain, meaning the total number of operations is proportional to the initial number of nodes. For instance, in a graph with **one hundred** nodes, the algorithm performs **ninety-eight** basic operations. The following figure demonstrates this relationship.

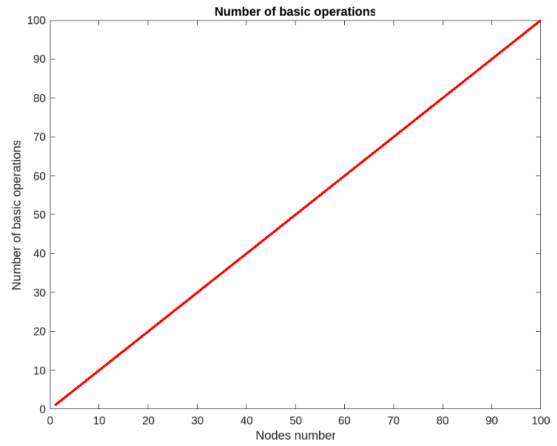

Fig. 6 - Number of basic operations

B. Execution Time

The execution time is closely correlated with the number of nodes and edges in the graph, with the number of edges being the most influential factor.

The figure above illustrates three key components: execution time, the number of nodes, and the number of edges in the graph. As the number of nodes, the execution time also rises. However, the number of edges is the primary variable driving significant increases in execution time. This is evident when comparing two graphs: if one has fewer nodes but more edges than the other, the execution time for the graph with more edges is notably longer.

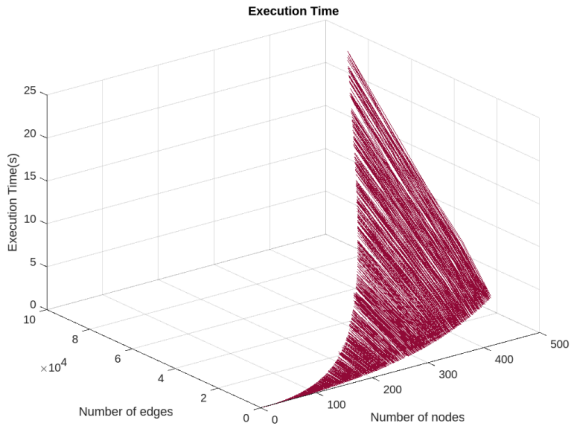

Fig. 7 - Execution time

C. Number of Solutions and Configurations Tested

This algorithm determines the minimum cost associated with each graph and identifies cut edges, providing a single solution for each graph. However, as the algorithm can sometimes yield incorrect results, each graph was tested **10 times**. The minimum value obtained from these trials is then recorded as the best solution for that graph. Based on the previously presented formula, it is recommended to run the algorithm $n^2 log(n)$ times for each graph minimize the probability of failure. However, this approach would be too time-consuming and was therefore not implemented. In this context, the probability of failure is defined by the following formula:

$$Pfail \leq (e^{-\frac{1}{n^2}})^{10} \leq e^{-\frac{10}{n^2}} \leq \frac{1}{e^{\frac{10}{n^2}}}$$

where, **n** represents the number of nodes. Using an example, for **n = 100** the **probability of failure** is:

$$Pfail \leq (e^{-\frac{1}{100^2}})^{10} \leq e^{-\frac{10}{100^2}} \leq \frac{1}{e^{\frac{1}{10}}} \leq 0.91$$

At first glance, this value might suggest that the algorithm has an accuracy of less than 10%, leading to the assumption that it is ineffective. However, what these values actually indicate is that the probability of consistently returning the correct result in all **10 executions** is **10%**. With this implementation, the algorithm only needs to find the correct solution once, as it ultimately selects the lowest cost as the final solution. Therefore, it can be inferred that the algorithm achieves a favorable balance between execution time and correctness.

This algorithm guarantees that each edge is only tested only once. Once an edge is selected, it is removed from the list of edges to be tested. Additionally, other edges are eliminated, such as those that would create loops.

D. Accuracy of the solutions obtained

To analyse the results, a comparison was made with the outcomes of an exhaustive search, which always finds a solution if one exists. Karger's algorithm successfully identifies the solution in approximately **30%** of cases, aligning with the previously started probability of success. Consequently, this algorithm does not always yield the optimal solution. This limitation can be attributed to the varying weights of the edges.

VI. Code Structure and Organization

In the submitted zip file, there exists a directory called "*src*" where inside exists all the code developed for this project, as shown in Fig. 7.

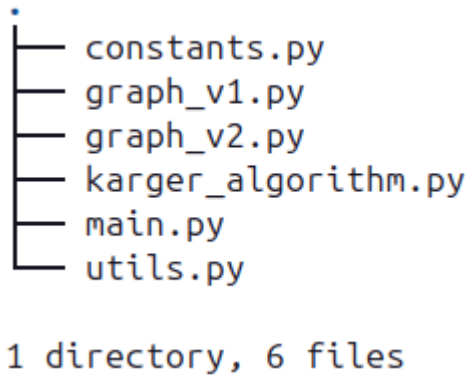

Fig. 8 - Folder organisation

The **constants.py** file contains three variables that have always the same value which are the **STUDENT_NUMBER**, the **EDGES_PERCENTAGE** which is a list containing all the edge percentages used to generate the graphs and **GRAPHS** which contains the number of graphs randomly generated.

The **graph_v1.py** contains a class called ***GraphV1*** which represents the graph developed in the first project, but with some changes.

The **graph_v2.py** has also a class called ***GraphV2*** that was developed to create, build and draw the graphs provided by the course teacher.

The **karger_algorithm.py** contains a class designated ***KargerAlgorithm*** containing the implementation developed for the Karger's algorithm.

The **main.py** is the entrypoint to execute the solution developed. It provides an argument when executing it, which is **a path to a folder that contains the graphs provided by the teacher**. If this argument isn't passed then the graphs will be randomly generated.

The **utils.py** contains two functions that depend on the folder option, mentioned above. If the argument is passed then the **use_teachers_graph** will be executed, using the ***GraphV2*** class, also mentioned above. As a first step it reads the first lines of the file where it's indicated if the graph is directed, the number of nodes and edges and the connections between the nodes and their associated weights. Then the karger's algorithm is applied for the graph, where its executed **10 times** and the results are written to a single file following this name structure:

Graph_<**number of the graph**>(Nodes_<**number of nodes**>, Edges_<**number of edges**>).txt

In the end, some general information is written, like the lowest value obtained for the cut, an average for the execution time and also an average for the number of basic operations performed. These files are stored in a directory, that is created with the name **solution**

If the folder option isn't provided, then the **generate_random_graphs** is executed where random graphs will be generated and the karger's algorithm will be applied to each one **10 times**. The results are then stored in text files, following the same naming structure as the one mentioned above, with the difference that these are stored in a directory called **random_graphs**.

VII. Conclusion

This work provided a deeper understanding of randomized algorithms, particularly **Karger's Algorithm** and its properties. Although it is designed for weighted graphs, it also performs well in this context. One notable aspect is how the algorithm's behavior changes with the increase in the number of nodes and edges. However, it is important to note that this algorithm isn't well-suited for the **Minimum Edge Cut Problem** due to its low probability of success.

References

[1] Peng Hui How, Virginia Williams, "Min Cut and Karger's Algorithm type", November 20, 2016
https://web.stanford.edu/class/archive/cs/cs161/cs161.1176/Lectures/CS161Lecture16.pdf

[2] Paul Learns Things, Karger's Algorithm: Procedure,
https://www.youtube.com/watch?v=KqMGeNZuwfI

[3] Wikipedia, "Karger's algorithm"
https://en.wikipedia.org/wiki/Karger's_algorithm

[4] Wikipedia, "Stoer-Wagner Algorithm"
https://en.wikipedia.org/wiki/Stoer%E2%80%93Wagner_algorithm

[5] GeekForGeeks, "Karger's algorithm for Minimum Cut"
https://www.geeksforgeeks.org/introduction-and-implementation-of-kargers-algorithm-for-minimum-cut/

[6] Top Coder, "Computational Complexity Part one"
https://www.topcoder.com/thrive/articles/Computational%20Complexity%20part%20one