Use of Exhaustive Search and Greedy Heuristics on the minimum cut graph problem

Pedro Ponte
Licenciatura em Engenahria 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, Exhaustic Search, Greedy Heuristic**

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

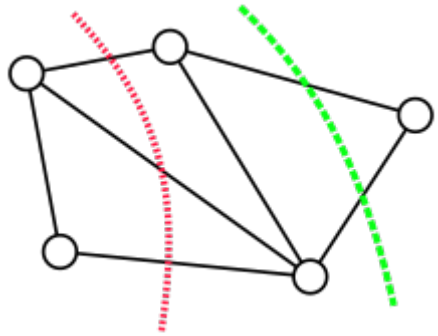

Fig. 1: Example with a cut of three edges represented by a dotted red line and a minimum cut represented by a dotted green line.

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. Exhaustive Search

An *Exhaustive Search* is based on a brute force approach where all possible solution candidates are evaluated until we find our desired result.

This search is

- Guaranteed to find a solution
- Easy to implement
- Inefficient for large inputs

which means our minimum cut problem will always have a correct solution, but as the graph size increases, the solution time also grows exponentially.

C. Greedy Heuristic

A *Greedy Heuristic Search* is a strategy that always follows the optimal choice at each step, doing so without considering the full development of the solution.

This search

- Never backtracks or revises decisions
- Usually faster than exhaustive search
- May not find the optimal solution

this will be a faster method than the exhaustive search, but won't result in the optimal solution.

II. Algorithms

A. Exhaustive Search

The *Exhaustive Search* algorithm explores every possible way of dividing the graph into two sets.
First, we'll use binary numbers to represent all possible partitions. For a graph with **n** nodes, it generates numbers from 1 to $2^{(n-1)} - 1$ in binary. For 5 nodes:

$$00000, 00001, 00010, ..., 11111$$

Each number represents a partition with two sets of nodes, where **00111** means nodes 0-1 belong to **set A** and nodes 2-4 belong to **set B**.
For each partition the algorithm:

- Creates the two sets of nodes
- Counts the number of edges between the two sets (**cut size**)
- Saves the partition with the smallest cut size

we count from 1 and use $2^{(n-1)} - 1$ instead of $2^{(n-1)}$ to avoid empty sets.
The algorithm returns

- The minimum cut size found
- The partition (the two sets of nodes) that achieved this minimum

With this we're guaranteed to have the optimal solution, since we check all possible partitions.

B. Greedy Heuristic

The search using a *Greedy Heuristic* tries to find the optimal solution locally, by using an heuristic to cal-

culate if the current cut should change or not.
We start with a simple partition, created by diving the nodes into two similar groups, where the first half of nodes go to *set A*, the other half goes to *set B* and then we count the number of edges between both sets. Now we enter the main loop, where for each vertex in the graph we'll:

- Check if moving a vertex from one set to ther other is allowed (doesn't leave the set emtpy)
- Calculate what we gain from moving it by
  - counting current crossing edges (edges that connect vertices from one set to the other)
  - counting current crossing edges
  - subtracting to get the improvement
- Keep track of the vertex that gives the best improvement

After checking all vertices we'll decide what to do, if there is no possible improvement the program stops, otherwise the best vertex found is moved to the other set.
Then we can update our best solution with the new cut and proceed to the next iteration.

III. Analysis

A. Formal Analysis

To better understand how our algorithms 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 them and also give us a way of comparing them in terms of performance.

A.1 Exhaustive Search

For the *exhaustive search* we can follow the instructions:

1. iterate through $2^{(n-1)}$ partitions - O(2^n)
2. for each partition
   - create sets - O(n)
   - count cut edges - O(n^2)

which results in

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

A.2 Greedy Heuristic

The same applies to the *greedy heuristic*:

1. go through the original partition - O(n)
2. for each iteration
   - check each vertex - O(n)
   - for each vertex
     - calculate gain - O(n)
     - make move and recalculate cut - O(n^2)

this is equal to

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

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

B. Experimental analysis

For this analysis, the graphs were created using the package **NetworkX**, specifically based on the method *powerlaw_cluster_graph*, using my NMec **98059** as a seed and . All code used for generation can be found in **utils.py**, while the plot generation can be found in **plot_generation.py**.

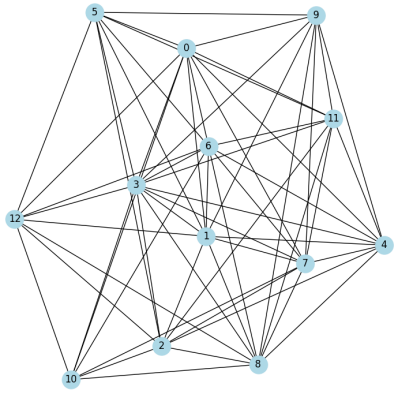

Fig. 2: Generated graph with 13 nodes

B.1 Operation count

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

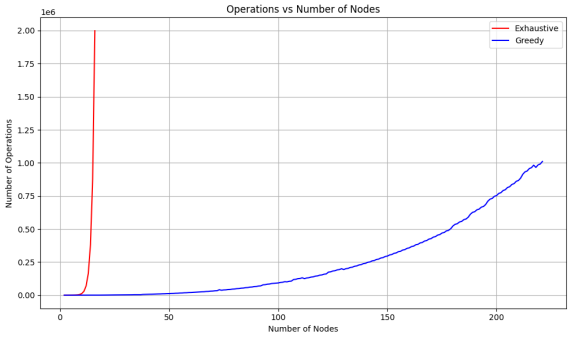

Fig. 3: Number of operations based on graph nodes with regular scale

from these plots we can confirm our formal conclusion that the **exhaustive search** grows a lot faster than the **greedy search**.

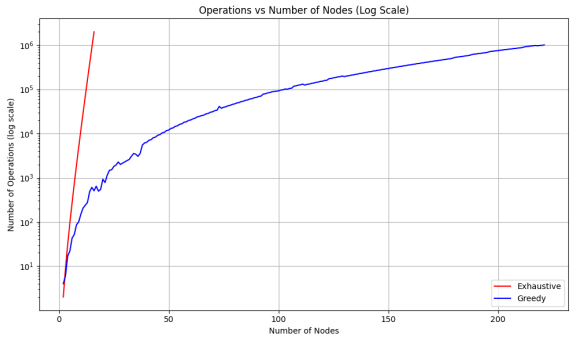

Fig. 4: Number of operations based on graph nodes with logarithmic scale

Additionally, the complexity is confirmed by looking at the logarithmic scale plots, where we can see an approximately straight line for the **exhaustive search**, which represents exponential grow, and a curved line for the **greedy search** which represents polynomial growth.

B.2 Time measurement

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

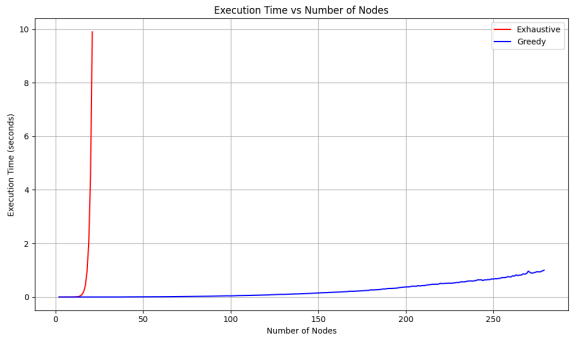

Fig. 5: Execution time based on graph nodes with regular scale

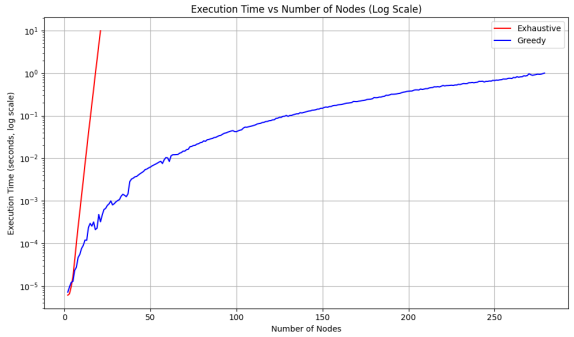

Fig. 6: Execution time based on graph nodes with logarithmic scale

Once again we can confirm that time grows as expected in the formal analysis.

Furthermore we can make some observations about the performance of both algorithms.
For the **exhaustive search**, we can get to 20 nodes before the execution time starts to become too large to compute

Number of Nodes	Time (seconds)
...	...
10	0.00
11	0.00
12	0.01
13	0.10
14	0.04
15	0.09
16	0.19
17	0.43
18	0.94
19	2.12
20	4.58
21	10.17

TABLE I: Execution time of exhaustive algorithm for different graph sizes

For the **greedy search** however, we can see that our execution does not follow the theoretical analysis.
As the number of nodes increases, so should the time, since our complexity is $O(n^4)$, but as we can see it stabilizes at around a 1.4x factor.
This result can be explained by memory caching effects and system-level efficiencies, like CPU pipelining and operating system scheduling, which optimizes the processes execution.

Number of Nodes	Time (seconds)
50	0.01
100	0.05
150	0.16
200	0.37
250	0.72
300	1.29
350	2.35
400	3.37
450	4.70
500	7.18
562	10.40

TABLE II: Execution time of greedy algorithm for different graph sizes

Node Range	Time Increase (s)	Factor Increase
50 → 100	0.04	5.0x
100 → 150	0.11	3.2x
150 → 200	0.21	2.3x
200 → 250	0.35	1.9x
250 → 300	0.57	1.8x
300 → 350	1.06	1.8x
350 → 400	1.02	1.4x
400 → 450	1.33	1.4x
450 → 500	2.48	1.5x
500 → 562	3.22	1.4x

TABLE III: Time increase analysis between node intervals

B.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 graphs it could handle with a reasonable time limit were:

- **Exhaustive Search**: 26 nodes -
- **Greedy Search**: 624 nodes - 15 seconds

B.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, with a particularly steep increase in weight towards the 0.75 probability rate

B.5 Heuristic precision

Unfortunately, due to the difference in execution from both algorithms and considering this problem may have multiple solutions for the same graph, it's hard to predict the accuracy of our heuristic, however it's possible to create an example where it fails

B.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**.

Nodes	Probability	Min Cut Weight
5	0.5	1
5	0.6	1
5	0.7	2
5	0.8	2
5	0.9	3
13	0.5	3
13	0.6	4
13	0.7	6
13	0.8	9
13	0.9	10
20	0.5	5
20	0.6	6
20	0.7	9
20	0.8	11
20	0.9	15

TABLE IV: Minimum cut weight for select graph sizes and edge probabilities (≥ 0.5)

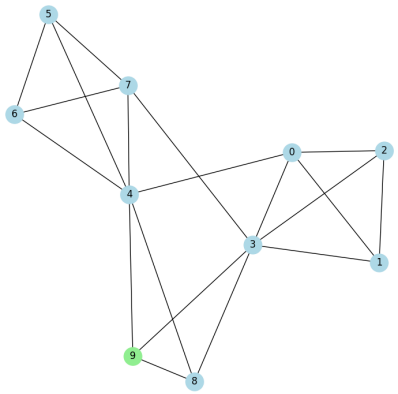

Fig. 7: Correct solution using the **Exhaustive Search**

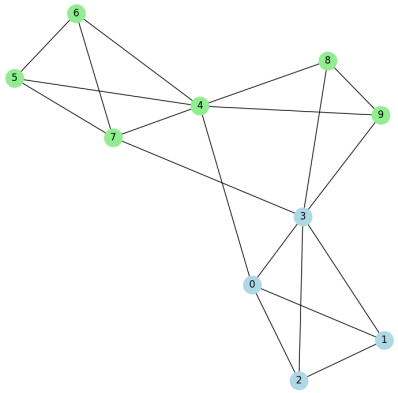

Fig. 8: Wrong solution using the **Greedy Heuristic**

IV. Extra results

Some computations of the **Exhaustive Search** algorithm for 3 different sizes.

TABLE V: Greedy Algorithm Predicted Execution Times

Nodes	Execution Time
100	0.26 seconds
500	6.66 seconds
1000	70.82 seconds (1.18 minutes)
5000	10817.43 seconds (3.01 hours)

TABLE VI: Exhaustive Algorithm Predicted Execution Times

Nodes	Execution Time
27	20.71 minutes
30	3.85 hours
35	8.91 days
40	495.46 days (1.36 years)

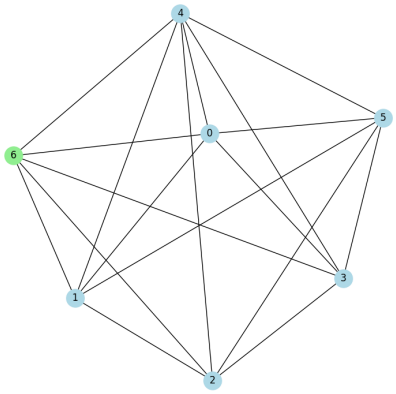

Fig. 9: Solution for graph with 7 nodes using **Exhaustive Search**

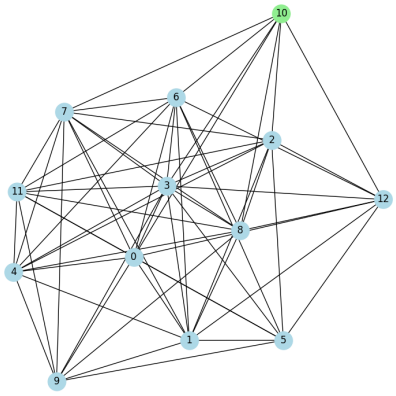

Fig. 10: Solution for graph with 13 nodes using **Exhaustive Search**

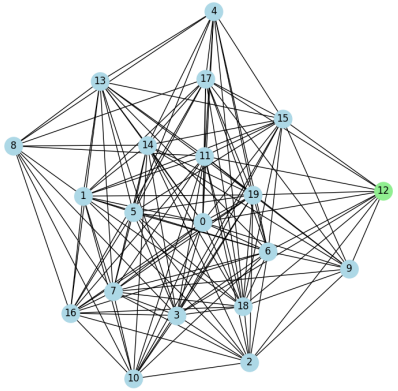

Fig. 11: Solution for graph with 20 nodes using **Exhaustive Search**

V. Conclusion

Our experimental analysis demonstrates the difference between the exhaustive and greedy approaches to the Minimum Cut problem. The exhaustive algorithm's execution time grows exponentially, becoming impractical beyond 27 nodes, with predicted times reaching 495 days for a 40 node graph. In contrast, the greedy heuristic shows significantly better scalability, handling graphs with 5000 nodes in approximately 3 hours.

These results clearly show why heuristic approaches are essential for practical applications, despite not guaranteeing optimal solutions. Future improvements could focus on balancing solution quality with computational efficiency for the greedy algorithm, particularly for larger graph instances.