Algoritmos Avançados 2024/2025

1º Semestre 1st Project

Exhaustive Search

Carlos Soto Torres

8 – For a given undirected graph G(V, E), with n vertices and m edges, does G have a cut with k edges? A cut of G with k edges 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 k.

In this exercise, I am working with two approaches to solve the graph cut problem: **exhaustive search** and a **greedy heuristic algorithm**. The **greedy heuristic** is based on the idea of making **local optimal choices** at each step. This means that at each step, the algorithm makes the decision that seems best at that particular moment, with the expectation that these local choices will lead to an overall good solution.

Let me explain the **greedy heuristic** that I implemented in my algorithm.

Greedy Algorithm

The greedy algorithm follows a strategy of **making local optimal choices**. At every step, it picks the "best" vertex or edge available based on a simple rule, without worrying about how this choice might affect future decisions.

In my case, the heuristic used is based on the **sum of the degrees of the vertices** that the edges in the graph connect.

Heuristic:

- **Edge Selection**: The greedy algorithm sorts all the edges in the graph by the sum of the degrees of the vertices involved in each edge. In simpler terms, for each edge (u,v)(u, v)(u,v), the algorithm calculates the **degree sum** of vertices uuu and vvv, which is:

Degree sum=degree(u)+degree(v)\text{Degree sum} = \text{degree}(u) + \text{degree}(v)Degree sum=degree(u)+degree(v)

The degree of a vertex is simply the number of edges connected to it.

- **Selecting Vertices**: The algorithm prioritizes edges that connect vertices with the highest degrees. This is because vertices with a higher degree are more connected, and by cutting edges involving these vertices, we may increase the number of edges between the two sets in the partition. The idea is that by focusing on high-degree vertices, I can potentially create a more impactful cut, which could result in a better solution.

Procedure:

1. **Sort the Edges**: First, the algorithm sorts all the edges by the sum of the degrees of the vertices they connect.
2. **Select an Edge**: Then, at each step, the algorithm picks an edge from the sorted list and assigns the vertices involved in the edge to one of the sets. If neither of the vertices is assigned yet, one of them is added to the partition.
3. **Form the Partition**: The algorithm continues this process, adding vertices to the partition and creating two subsets SSS and TTT of the graph.

4. **End**: The algorithm stops when no more vertices can be added to the partition, or when a complete partition is formed.

Why Did I Use This Heuristic?

- **Simplicity**: It's a straightforward approach that doesn't require examining all possible partitions, as would be the case with exhaustive search.
- **Efficiency**: Since the algorithm only focuses on edges involving vertices with high degrees, it’s faster than exploring all combinations.
- **Approximation**: While the algorithm doesn't guarantee an optimal cut, it often provides a good solution, especially since it tends to focus on the most "important" edges in the graph.

What Is This Heuristic Trying to Achieve?

The aim of this heuristic is to find a way to partition the graph into two sets, SSS and TTT, such that the number of edges between them is large. By using the degree-based approach, the algorithm aims to create a cut that maximizes this edge count, without the computational expense of considering all possible partitions.

Following the points:

a) Perform a formal computational complexity analysis of the developed algorithms

For the **exhaustive search algorithm**, it is a **search across all possible partitions** of the graph. This means that the algorithm checks every combination to find the optimal cut. The complexity of this approach is:

- **Combinatorial**: In a graph with nnn vertices, the number of possible partitions is exponential. In the worst case, if we have to try all possible partitions, the complexity would be O(2n)O(2^n)O(2n), as each vertex can be assigned to either set SSS or set TTT.

For the **greedy algorithm**, the complexity is determined by the operations performed to sort the edges and then process them. If we have EEE edges and VVV vertices, the complexity would be:

- **Sorting the edges**: This is done in O(ElogE)O(E \log E)O(ElogE) using an efficient sorting algorithm.
- **Partition processing**: After sorting, processing each edge takes O(E)O(E)O(E).

Together, the total complexity of the greedy algorithm is O(ElogE)O(E \log E)O(ElogE), which is much more efficient than exhaustive search.

---

b) Conduct a sequence of experiments

1. **Number of basic operations**: To record the number of basic operations, I can count the number of iterations or comparisons performed during the algorithm's execution. In the exhaustive search, it will be the number of vertex combinations tested. For the greedy algorithm, it will be the number of edges processed.
2. **Execution time**: To measure the execution time, I will use tools in Python, like the time module, to calculate the total time it takes for each algorithm to run.
3. **Number of solutions/configurations tested**: In the case of exhaustive search, the number of configurations tested will be equal to the number of possible graph partitions. For the greedy algorithm, the number of configurations tested is equal to the number of edges processed.
4. **Greedy heuristic accuracy**: The accuracy of the heuristic can be compared with the optimal result from the exhaustive search. If the difference between the cut obtained by the greedy algorithm and the optimal cut is small, then we can consider the heuristic to be accurate.

---

c) Compare the results of the experimental and formal analysis

- In the **formal analysis**, I have established the complexities of both algorithms (exhaustive and greedy) in terms of how they behave with the number of vertices and edges.

- In the **experimental analysis**, through the experiments, I will obtain data on the execution time and the number of operations performed. By comparing both analyses, I can validate the theory and observe whether the theoretical complexity matches the actual performance.

---

d) Determine the largest graph that you can process on your computer without taking too much time

To determine the **largest graph** that can be processed without excessive computation time, I will progressively increase the size of the graphs in my experiments and measure the execution times until they become too large or the algorithm cannot finish within a reasonable time. Depending on my computer's resources, there will be a point where the graph size exceeds what can be processed efficiently.

---

e) Estimate the execution time for much larger problem instances

Based on the results obtained in my experiments, I can make an extrapolation of the execution times to estimate how long it would take to run much larger instances. I can do this by fitting the results to a **growth curve** and using mathematical models to approximate the time it would take for a much larger number of vertices.

Examples:

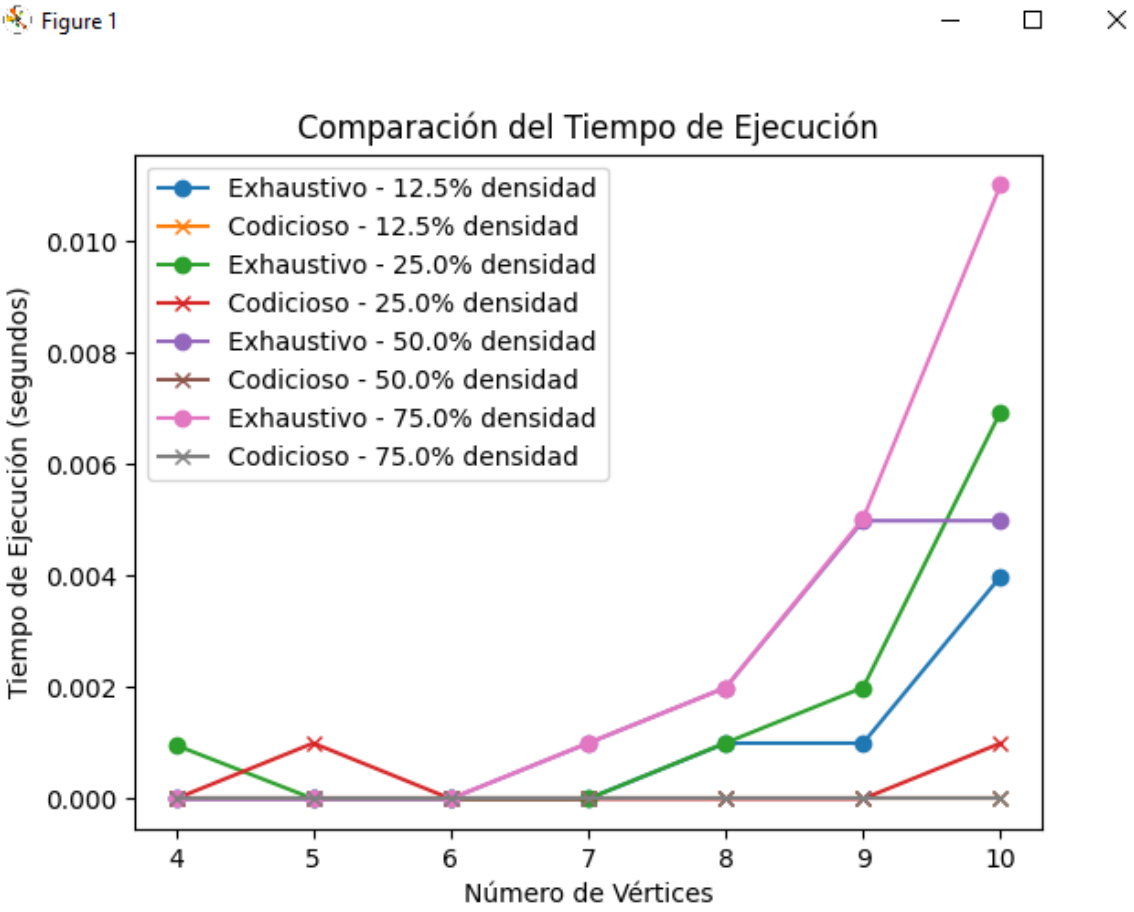
Figure 1
Comparación del Tiempo de Ejecución
Exhaustivo - 12.5% densidad
Codicioso - 12.5% densidad
Exhaustivo - 25.0% densidad
Codicioso - 25.0% densidad
Exhaustivo - 50.0% densidad
Codicioso - 50.0% densidad
Exhaustivo - 75.0% densidad
Codicioso - 75.0% densidad
Tiempo de Ejecución (segundos)
0.000
0.002
0.004
0.006
0.008
0.010
4
5
6
7
8
9
10
Número de Vértices

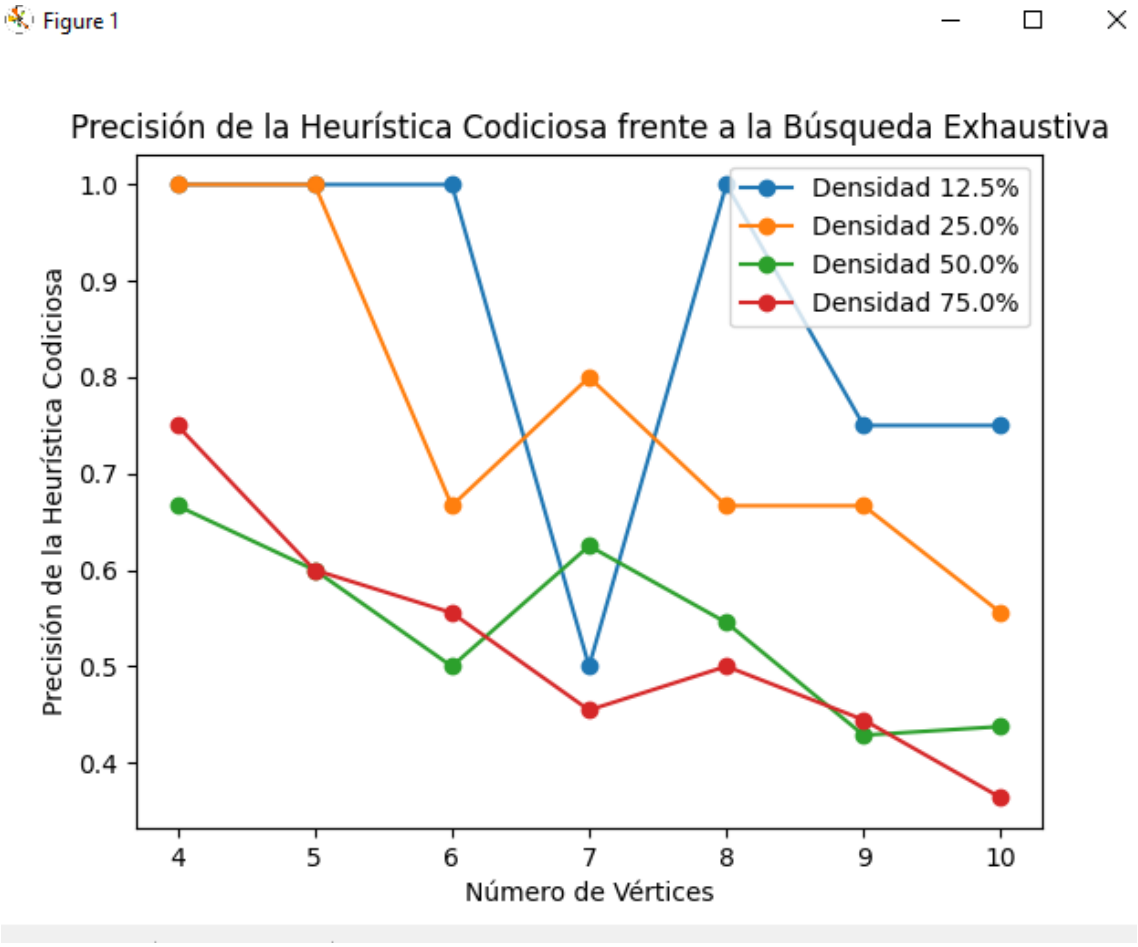
Figure 1
Precisión de la Heurística Codiciosa frente a la Búsqueda Exhaustiva
Densidad 12.5%
Densidad 25.0%
Densidad 50.0%
Densidad 75.0%
Precisión de la Heurística Codiciosa
1.0
0.9
0.8
0.7
0.6
0.5
0.4
4
5
6
7
8
9
10
Número de Vértices

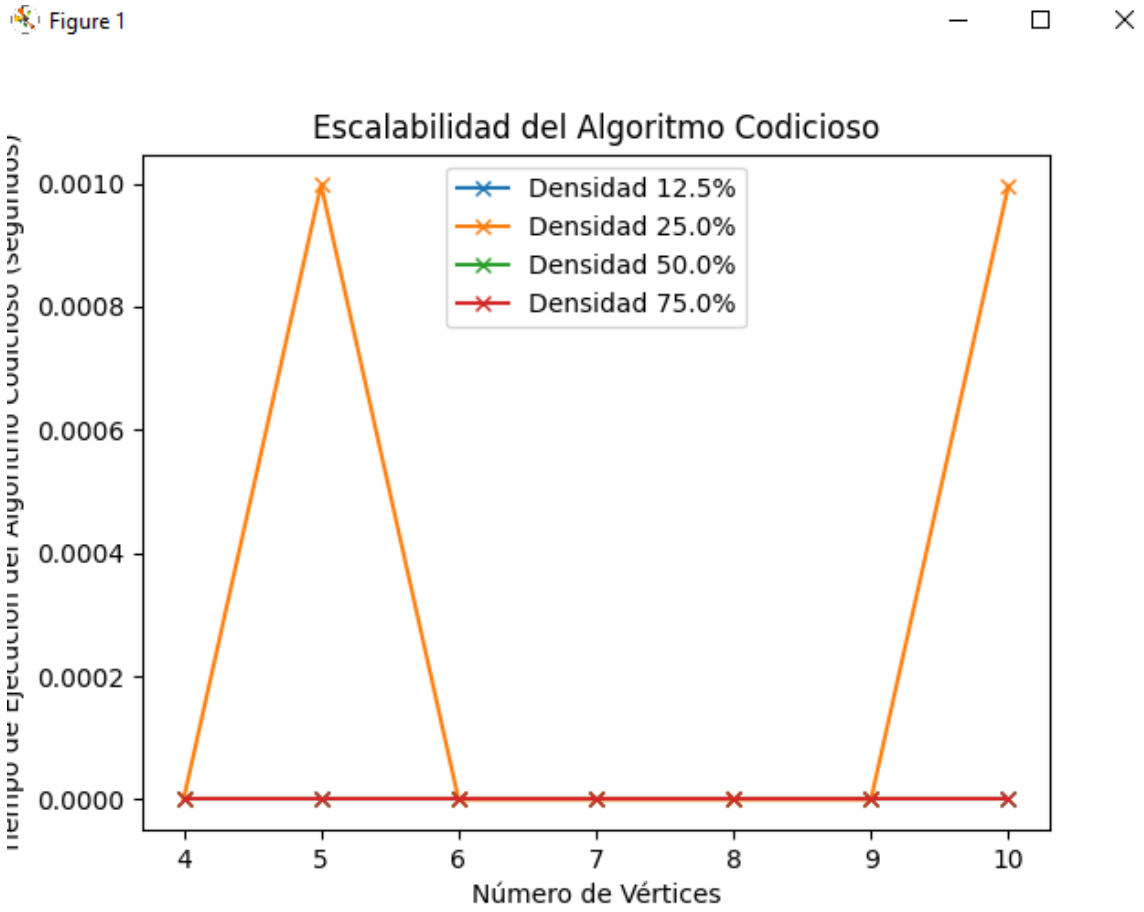
Figure 1
Escalabilidad del Algoritmo Codicioso
Densidad 12.5%
Densidad 25.0%
Densidad 50.0%
Densidad 75.0%
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
4
5
6
7
8
9
10
Número de Vértices