Subgraph Isomorphism: Bonnici-Giugno Algorithm

Arjuna Herbst
CPSC 450
Fall 2024

Summary

The subgraph isomorphism problem is a computational task in which two graphs G and H are given as input, and the challenge is to determine whether G contains a subgraph that is isomorphic to H. Using Python, I was able to implement two approaches to solving this problem: a naïve backtracking approach, and the Bonnici-Giugno (RI) algorithm. After running performance tests involving calculating the average runtime over a variety of different graphs, The Bonnici-Giugno algorithm was found to outperform the naïve approach exponentially in terms of runtime.

1. ALGORITHM SELECTED

In the subgraph isomorphism problem, two graphs G and H are given as input. Graphs G and H are isomorphic if there is a structure that preserves a one-to-one correspondence between the vertices and edges. Since subgraph isomorphism is a generalization of both the maximum clique problem as well as the problem of testing whether a graph contains a Hamiltonian cycle, it is NP-Complete. Finding a solution involves finding a mapping of nodes between H and G such that the edges in H are preserved in G.

My naïve approach generates all possible permutations of nodes in G taken k at a time (where k is the number of nodes in H). Each permutation represents a potential mapping of H's nodes to G's nodes. For each generated permutation, the algorithm constructs a mapping between the nodes of H and the nodes in the permutation. Then, it checks whether for every edge in H, the corresponding edge exists in G according to the node mapping. If this condition is true, a valid isomorphism is found. This algorithm is very inefficient, with a time complexity of $O(n^k \cdot m)$ where m is the number of edges in H. The pseudocode for my naïve approach can be found below.

```
1. if $|V_H| > |V_G|$:
2.     return None
3. for each permutation $P$ of size $|V_H|$ from $V_G$:
4.     create a candidate mapping $M: V_H \rightarrow V_G$
5.     if $\forall(u, v) \in E_H, (M(u), M(v)) \in E_G$:
6.         return $M$
7. return None
```

On the other hand, my implementation of the Bonnici-Giugno algorithm uses the Greatest Constraint First algorithm, which is much more efficient than naïve backtracking. The concept is to start with the vertex in H that has the most constraints. This step reduces the search space by trying to match the more constrained vertices first, which leads to less backtracking steps. [1] This algorithm starts at the vertex in H of maximum degree, then iteratively adds the next vertex based on the following: Number of edges to already ordered vertices, number of edges to remaining vertices, and vertex degree. From here, the Bonnici-Giugno algorithm then uses recursive matching to attempt to find a valid mapping of the ordered vertices from H to the nodes in G. It then checks if the neighbors of the current vertex in H have already been mapped to neighbors in G. If a valid mapping is found for the current vertex, the algorithm recurses to match the next vertex, on failure it backtracks by removing the most recently added node and trying another potential node. [1] While this algorithm has a similar time complexity of $O(k^2 + n^k)$, it is faster than the naïve approach due to the pruning enabled by GCF ordering. The pseudocode for the Bonnici-Giugno algorithm can be found below.

```
1. if $|V_H| > |V_G|$ or: $|V_H| = 0$ or $|V_G| = 0$:
2.     return None
3. orderedPattern $\leftarrow$ greatesConstraintFirst($H$)
4. $M \leftarrow$ empty mapping
5. matchRecursive(orderedPattern, $M$, {})
```

2. IMPLEMENTATION

The algorithms used in this project were implemented in Python using the Networkx library to handle graph representations and operations such as accessing nodes, edges, degrees and neighbors.

My naïve backtracking approach uses NetworkX Graph objects for graph representation. It uses the itertools libraries permutations method to generate node mappings, and Python dictionaries for mapping validation. This algorithm is impractical for large graphs due to its factorial growth in number of permutations.

My implementation of the Bonnici-Giugno algorithm uses the greatest constraint first as its heuristic. The implementation also uses NetworkX Graph objects for graph representation, Python lists to store the order nodes of H, Python dictionaries for node mappings, and Python sets to track matched vertices. Implementing the heuristic involved balancing factors such as edge degrees and connectivity, and the recursive backtracking process involved a lot of debugging to handle edge cases.

To ensure the correctness of each implementation, I created a suite of unit tests that covers a variety of graph configurations and scenarios. Each test case was designed to test a specific aspect of each algorithm.

test_empty_graphs: Validate algorithm behavior when either G or H are empty. This ensures that both algorithms exit correctly when an empty graph is given as input.

test_simple_isomorphism: Validate that both algorithms can identify isomorphism for a straightforward subgraph problem.

test_no_isomorphism: Validate that both algorithms can identify when there is no isomorphism.

test_single_node_subgraph: Validate that both algorithms can properly handle subgraphs with only 1 node.

test_identical_graphs: Validate that both algorithms are able to identify isomorphism when *G* and *H* are identical.

test_partial_overlap: Validate that both algorithms can identify that no isomorphism exists when *H* is 1 edge off being a subgraph of *G*.

test_disconnected_subgraph: Validate that both algorithms can identify isomorphism of disconnected subgraphs within a larger graph

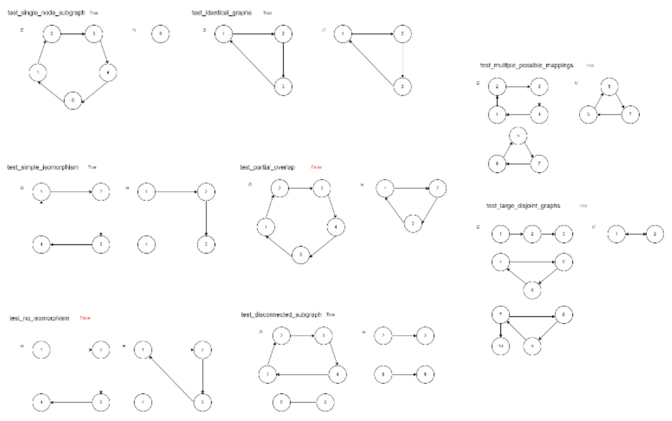
test_large_isomorphic_subgraph: Validate that both algorithms can identify isomorphism with sizes of *G*=20, *H*=10

test_multiple_possible_mappings: Validate that both algorithms can find isomorphism when more than 1 mapping exists between *G* and *H*.

test_large_disjoint_graphs: Validate that both algorithms can identify isomorphism in large graphs with disjoint components.

Diagrams of some of the notable graphs used are found below, with the title matching the test case it was used for.

Figure 1. Unit Test Graphs


test_single_node_subgraph
test_identical_graphs
test_multiple_possible_mappings
test_simple_isomorphism
test_partial_overlap
test_large_disjoint_graphs
test_no_isomorphism
test_disconnected_subgraph

3. PERFORMANCE TESTS

My first test framework scaled data inputs by increasing the size of the graphs. Each test configuration was designed to evaluate both the naïve approach and Bonnici-Giugno algorithm under the same conditions and can be found below in Table 1. Execution times were measured using the Python time library to compare efficiency.

Table 1. *G* Configurations

Size	Density
5	0.4
10	0.4
15	0.4
20	0.4
25	0.4
30	0.4
35	0.4
40	0.4

My second test framework followed a similar approach, however this time the size of subgraph *H* was increased, while *G* was kept at a constant configuration of size 13 with density 0.3. *H* was created with size from 5-12 for this test, since when $|V_H| > |V_G|$, both algorithms will return 0.

These tests were run on a Dell XPS 15 9500 with an Intel(R) Core(TM) i7-10875H CPU 2.30 GHz, 32GB RAM, x64-based processor running Windows. To run the tests locally, first create a new Python virtual environment with:

```
python -m venv venv
```

Activate the environment with:

`venv/Scripts/activate` on Windows

`source venv/bin/activate` on Mac

Then to install dependencies run:

```
pip install -r requirements.txt
```

Now to run the performance tests:

```
python plot_performance_diff_graph_types.py
python plot_performance_diff_H_size.py
```

4. EVALUATION RESULTS

To analyze the results, lets take a look at the different graphs that were generated from my performance tests.

Figure 2. Performance Testing Results – Diff. Graph Types

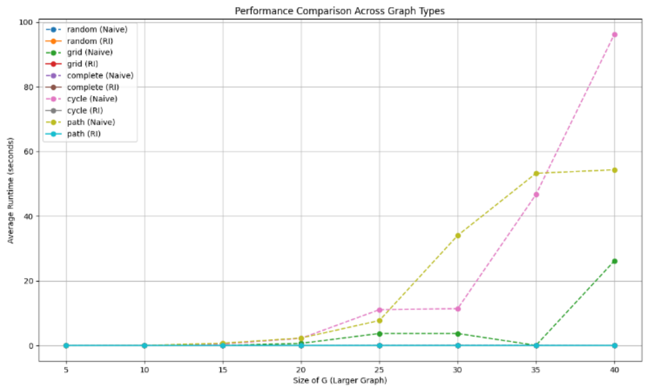


Figure 2 shows the performance of both approaches across various graph types, with runtimes measured for increasing sizes of *G*. Each graph type (random, grid, complete, cycle, path) is represented with separate lines for both algorithms. As the size of *G* increases, the Naïve algorithm's runtime grows exponentially, especially for dense graph types. In contrast, the Bonnici-Giugno algorithm shows better scalability, maintaining lower runtimes across all graph types and sizes. You can't see a few of the lines from the key because they are all overlapping with an average runtime of 0.0s.

The naive algorithm struggles with dense graphs because it relies on a brute force permutation based approach, leading to a massive search space. The more nodes and edges there are in *G*, the more permutations the naïve algorithm will have to check, leading to exponential growth in runtime as the size of *G* increases.

Figure 3. Performance Testing Results – Diff. Subgraph Sizes

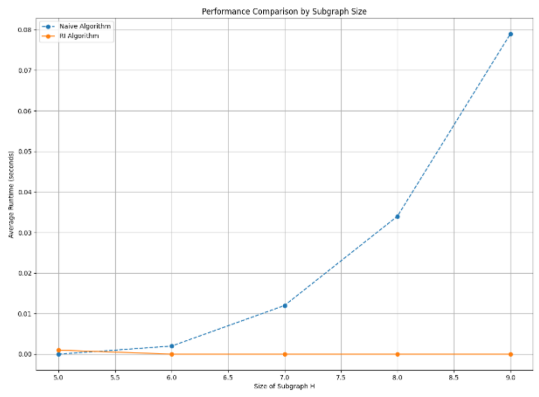


Figure 3 shows yet again exponential growth in runtime for the naïve backtracking algorithm as the size of the subgraph *H* increases. This makes sense with the results shown in Figure 2, as the naïve algorithm will have to create more permutations to find a valid mapping between *H* and *G* as the number of nodes increases in either graph.

5. REFLECTION

It was found that the Bonnici-Giugno algorithm is indeed an upgrade from a naïve brute force approach to solving the problem of subgraph isomorphism. It clearly outperformed the brute force algorithm for every performance test I implemented, varying sizes of *G* and *H*. This is mainly thanks to the greedy algorithm called GreatestConstraintFirst to find a good sequence of vertices. [1] This algorithm reduces the search space of valid mappings between *G* and *H*, allowing the Bonnici-Giugno algorithm to be very efficient with larger graphs, and I found it to be the most interesting aspect of the algorithm itself.

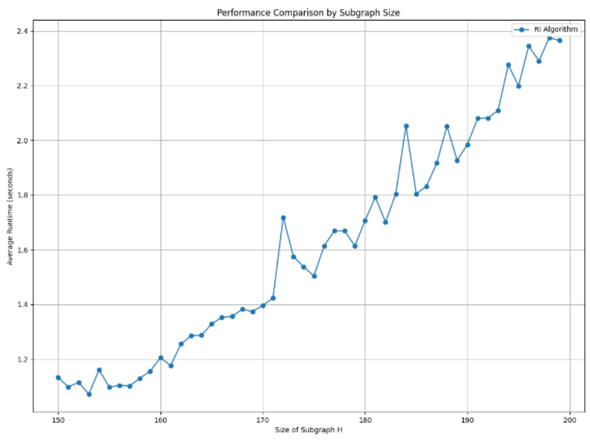
When researching how to implement the Bonnici-Giugno algorithm, I found that it is very similar to Depth-First Search (DFS). Both algorithms explore a graph, visiting nodes and following edges. Bonnici-Giugno algorithm succeeds in solving the subgraph isomorphism problem by creating a search space tree that represents valid mappings. Similarly, DFS explores a graph by following a path as far as possible before backtracking to explore other branches. [5]

As for challenges faced, I had originally planned to implement a test framework that scaled data inputs by increasing the size and density of the graphs. These were created using the nx.gnp_random_graph() method from the NetworkX library, which generates graphs based on the number of nodes (size) and edge probability (density). Graph density quantifies how many edges are present in a graph relative to the maximum number of edges possible between the vertices. For undirected graphs, it is calculated using the formula $D = (2*|E|)/(|V|(|V| - 1))$. [2] Each test configuration was designed to evaluate both the naïve approach and Bonnici-Giugno algorithm under the same conditions. However, I ran these performance tests overnight with no success. One pass of the naïve approach on a graph with size 20 and density 0.5 took around 10.5 hours to complete, and that was only the third of eight graphs to be tested. Due to this massive runtime, I chose to approach performance testing in a different manner.

Due to the fact that my naïve algorithm is so slow with large graphs, It was hard to properly gauge the performance of the Bonnici-Giugno algorithm. I was interested in how the Bonnici-Giugno algorithm would handle larger graphs, so I created another performance testing script, this one only testing the more efficient algorithm with *G* with size 2,000 & density 0.3, *H* with sizes 150-200. The results are shown below in Figure 4.

Next steps for this project would include expanded performance testing on the Bonnici-Giugno algorithm to begin. Transferring the codebase to a faster programming language such as C++ would also be beneficial, as it would allow for more extensive testing of both algorithms.

Figure 4.



6. RESOURCES

[3] Defines the subgraph isomorphism problem as a task in which two graphs *G* and *H* are given as input, and the challenge is to determine whether *G* contains a subgraph that is isomorphic to *H*. There are multiple ways to approach solving this problem, and I chose the methods from [5] and [1].

[5] Provided a good understanding of how a brute force approach might look. The author of this article included his implementation of a brute force approach to solving the subgraph isomorphism problem, and I took that code and tweaked it to work with the NetworkX library. This source also gives the time complexity of a brute force approach, which helped me understand my algorithm better.

[1] Proposes a new subgraph isomorphism algorithm which applies a search strategy to reduce the search space (Bonnici-Giugno algorithm). This algorithm extends the approach in [5] by using the Greatest Constraint First algorithm to reduce backtracking steps.

[2] Provides a definition of density in graphs, which I needed to understand in order to implement my performance tests with the NetworkX library.

7. REFERENCES

[1] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2012. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14, Suppl. 7 (May 2012), S13. https://doi.org/10.1186/1471-2105-14-S7-S13

[2] Omar Lizardo. Density. Retrieved December 6, 2024, from https://bookdown.org/omarlizardo/_main/2-9-density.html

[3] Wikipedia. 2024. Subgraph isomorphism problem. https://en.wikipedia.org/wiki/Subgraph_isomorphism_problem.

[4] S. Bowers, Fall 2024. Lecture 14: Depth-First Search. from https://www.cs.gonzaga.edu/faculty/bowers/courses/cpsc450/lect-14.pdf

[5] Toni Cañada. 2023. Brute-force code for isomorphisms. from https://tonicanada.medium.com/brute-force-code-for-isomorphisms-1241ef180570.