Algorithms and computability report

Erdni Mankirov
Darya Staliarova
Illia Komlichenko

December 2023

Contents

1 Descriptions of the concepts **2**
- 1.1 Size of the Multigraphs . . . 2
- 1.2 Distance between the Multigraphs . . . 2
  - 1.2.1 Example of Hamming Distance Calculation . . . 3
  - 1.2.2 Algorithms and computational complexity of the distance calculation and isomorphism test . . . 3
  - 1.2.3 Tests . . . 5
- 1.3 Maximum Clique in Multigraphs . . . 6
  - 1.3.1 Bron–Kerbosch Algorithm . . . 6
  - 1.3.2 Approximate Maximum Clique Algorithm . . . 7
  - 1.3.3 Example of the Maximum clique output . . . 8
  - 1.3.4 Some other tests . . . 9
- 1.4 Maximum common subgraph of two (or more) given multigraphs 10
  - 1.4.1 Description of the algorithm . . . 10
  - 1.4.2 Time Complexity Analysis: . . . 11
  - 1.4.3 Test . . . 11

2 Short Technical description **13**
- 2.1 Language and libraries . . . 13
- 2.2 Description of the GUI . . . 13
- 2.3 How to compile the program . . . 13
- 2.4 How to run the program . . . 14

3 Conclusion **15**

Chapter 1

Descriptions of the concepts

1.1 Size of the Multigraphs

The size of a multigraph is defined as the sum of its vertices and edges. In the context of multigraphs, the size is a measure that encapsulates both the number of vertices and the number of edges, providing a comprehensive indication of the overall complexity and structure of the multigraph.

For a given multigraph G, the size (size(G)) is calculated as follows:

$$\text{size}(G) = \text{number of vertices} + \text{number of edges}$$

This definition accounts for the contribution of both vertices and edges, emphasizing the combined impact of these elements on the overall size of the multigraph.

Understanding the size is crucial in analyzing and comparing multigraphs, as it provides insights into their overall scale and connectivity. In the subsequent sections, we explore various aspects of the multigraphs, including their largest cliques, adjacency matrices, and Hamming distances, to gain a comprehensive understanding of their structural characteristics.

1.2 Distance between the Multigraphs

The distance between two multigraphs is measured using the Hamming distance metric. The Hamming distance quantifies the dissimilarity between two multigraphs by counting the number of differing elements in their adjacency matrices. When comparing multigraphs with a different number of vertices, the distance is defined by extending the smaller matrix with zero rows and columns to match the size of the larger matrix.

For two multigraphs G_1 and G_2 represented by their adjacency matrices M_1 and M_2, the Hamming distance is given by:

$$\text{Hamming distance}(G_1, G_2) = \sum_{i,j} |M_1[i,j] - M_2[i,j]|$$

If the multigraphs are isomorphic, meaning they share the same structure (regardless of vertex labeling), the Hamming distance is 0.

1.2.1 Example of Hamming Distance Calculation

Consider the following multigraphs represented by their adjacency matrices:

$$G_1 = \begin{bmatrix} 0 & 3 & 0 \\ 3 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

and

$$G_2 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

To calculate the Hamming distance, knowing that the graphs are symmetric by main diagonal we take it's triangular matrix (L or U doesn't matter) and compare the differences in each element and count it:

$$\text{Hamming distance}(G_1, G_2) = \text{Hamming distance}(030010, 110010) = 2$$

The Hamming distance between the two multigraphs is 2, indicating that there are two differing elements on the triangular matrices.

1.2.2 Algorithms and computational complexity of the distance calculation and isomorphism test

The algorithm for calculating the Hamming distance between two graphs has a time complexity of $O(n^2)$. This complexity arises from the need to traverse the adjacency matrices of the graphs and compare corresponding elements. So firstly we should check if the graphs are NOT isomorphic. It is obvious when they different number of edges or vertices graphs are definitely not isomorphic and computation of their distance will equal $O(n^2)$.
Otherwise we will use VF2 algorithms to determine the isomorphism of the graphs.
VF2 Algorithm Description:

1. **Initialization:**
    - The VF2 algorithm begins by initializing its state. It takes as input two graphs, Graph1 and Graph2, and starts the isomorphism search.

2. **State Space Exploration:**

   - The algorithm explores the state space by iteratively considering pairs of vertices from the two graphs and attempting to match them. At each step, it maintains a current state, representing a partial mapping between vertices of Graph1 and Graph2.

3. **State Transition:**

   - The algorithm transitions between states by considering possible mappings for the next pair of vertices. It enforces conditions to ensure that the partial mapping is consistent with the graph structures.

4. **Feasibility Checks:**

   - At each step, the algorithm performs feasibility checks to verify if the current partial mapping can be extended to a full isomorphism. These checks include verifying adjacency relationships, checking for one-to-one correspondence, and ensuring that the mapping does not violate any constraints.

5. **Backtracking:**

   - If the algorithm reaches a point where it cannot extend the current mapping further without violating constraints, it backtracks to the previous state and explores alternative mappings. Backtracking is a key component that allows the algorithm to explore different branches of the state space.

6. **Isomorphism Detection:**

   - The algorithm continues exploring the state space until a valid isomorphism is found or all possibilities are exhausted. If a valid isomorphism is found, the algorithm returns a positive result, indicating that the two graphs are isomorphic.

The complexity for determining graph isomorphism can vary. In the worst-case scenario, the time complexity can be expressed as $O(n! \cdot n)$. This arises from the combinatorial nature of the isomorphism problem, and the factorial term accounts for the permutations of vertices. It's important to note that this worst-case complexity may be impractical for large graphs.

In more favorable scenarios, certain approximations of VF2 algorithm can approximate isomorphism with a time complexity of $O(n^2)$. However, the applicability of these algorithms may be limited to specific cases and may not guarantee accuracy in all situations.

In case of the file input with one graph program will return nothing.

1.2.3 Tests

The input Graphs

$$G_3 = \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix}$$

and

$$G_2 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Output: Graph 2 and Graph 3 are not isomorphic. Hamming Distance: 4. Calculation Time: 1.00 ms

$$\text{Hamming distance}(G_2, G_3) = \text{Hamming distance}(022020, 110010) = 4$$

Test 1 is correct. Proceed to the next test:

The input Graphs

$$G_4 = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

and

$$G_5 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

Output: Graph 4 and Graph 5 are isomorphic. Hamming Distance: 0. Calculation Time: 7.00 ms
Indeed they are isomorphic. So the distance is 0. Also we should pay attention that calculation time is 7 times higher than in previous non isomorphic case.
Test 2 is correct. Proceed for the next test:
The input Graphs

$$G_6 = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 0 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

and

$$G_7 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

and

$$G_8 = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 \end{bmatrix}$$

Output:
Graph 6 and Graph 7 are not isomorphic. Hamming Distance: 4. Calculation Time: 1.00 ms
Graph 6 and Graph 8 are not isomorphic. Hamming Distance: 6. Calculation Time: 1.00 ms
Graph 7 and Graph 8 are not isomorphic. Hamming Distance: 2. Calculation Time: 0.00 ms
Let's check it:
Firstly we extend the smaller matrix with zero rows and columns:

$$G_6 = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Then calculate a Hamming + the difference between the number of vertex (technically shapes of the matrices):

$$\text{Hamming distance}(G_6, G_7) = \text{Hamming distance}(0200020000, 0100010020) = 3$$

$$+1 \text{ vertexasdiffernce} = 4$$

$$\text{Hamming distance}(G_6, G_8) = \text{Hamming distance}(0200020000, 1101010020) = 5$$

$$+ 1 \text{ vertexasdiffernce} = 6$$

$$\text{Hamming distance}(G_7, G_8) = \text{Hamming distance}(0100010020, 1101010020) = 2$$

Test 3 is correct.

1.3 Maximum Clique in Multigraphs

In this section, we explore the concept of the maximum clique in a multigraph, defining it as the clique with the highest sum of its vertices and edges. We employ a slightly modified version of the Bron–Kerbosch algorithm to identify and calculate the size of the maximum clique.

1.3.1 Bron–Kerbosch Algorithm

The Bron–Kerbosch algorithm is a well-known algorithm for finding all cliques in an undirected graph. It was originally designed for simple graphs, but we

have adapted it to handle multigraphs by considering edges as multiplicities rather than binary relationships.

The modified Bron–Kerbosch algorithm follows these steps:

1. **Initialization:** - Start with an empty set representing the current clique and two sets representing potential candidates for the clique (P and X).

2. **Exploration:** - Choose a vertex from the set of potential candidates (P) and add it to the current clique. - Update the sets P and X based on the neighbors of the chosen vertex.

3. **Recursion:** - Recursively explore the remaining vertices in the updated sets P and X, adding them to the current clique.

4. **Backtracking:** - If the recursion reaches a point where no more vertices can be added without violating the clique property, backtrack to the previous state.

5. **Cliques Collection:** - Collect and record all cliques encountered during the exploration.

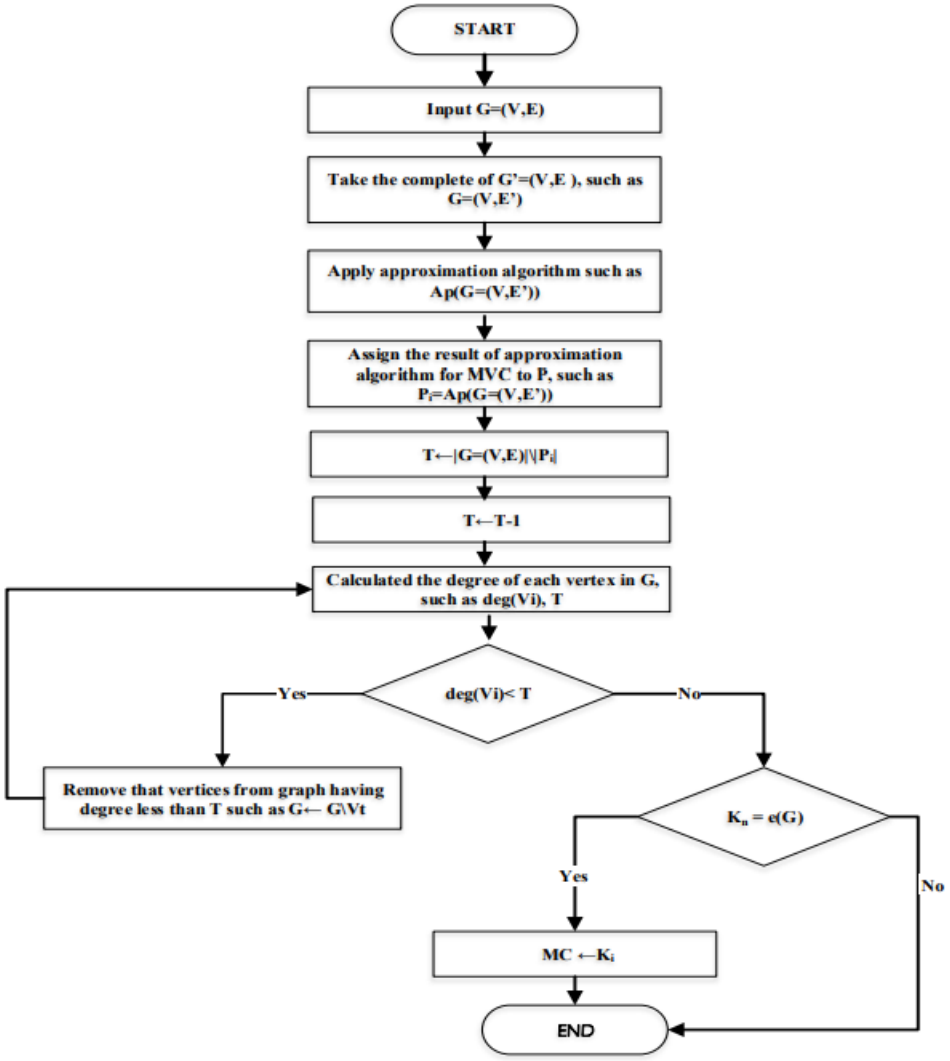
The complexity of the modified Bron–Kerbosch algorithm depends on the characteristics of the multigraph. In the worst case, the algorithm has an $O(3^{n/3})$ time complexity, making it less suitable for very large or dense multigraphs. However, its performance may be reasonable for smaller or sparser multigraphs.

The modified algorithm extends the original Bron–Kerbosch approach to account for multiplicities in edges, allowing it to handle multigraphs effectively.

Understanding the complexities of the algorithms employed is essential for evaluating their performance in the context of different multigraph structures and sizes.

1.3.2 Approximate Maximum Clique Algorithm

However the the Maximum clique problem is still NP , there is exists an algorithm that will allow us to calculate maximum clique in approximately polynomial time complexity. It's description stated on the next page.



1.3.3 Example of the Maximum clique output

Let's put as an input adjacency matrix

$$G_9 = \begin{bmatrix} 0 & 10 & 1 & 0 & 0 & 0 & 0 \\ 10 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

This graph represent a separated triangle and a square where all vertices are connected with each other. The algorithm for ordinary graphs would return as obviously a square as it has more vertices. But our program return this result:
Largest clique size:15
Adjacency Matrix:

$$\begin{bmatrix} 0 & 10 & 1 \\ 10 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Calculation Time: 146.96 ms
As in our multigraph this subgraph has 3 vertices and 12 edges while the square has 4 vertices and 6 edges. The size was defined as sum of vertices and the edges so as you can see the clique with less vertices still has bigger size.

1.3.4 Some other tests

Input Graph 1 :

$$G_1 = \begin{bmatrix} 0 & 3 & 0 \\ 3 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Largest clique size: 5
Adjacency Matrix:

$$\begin{bmatrix} 0 & 3 \\ 3 & 0 \end{bmatrix}$$

Calculation Time: 72.02 ms
Input Graph 3 :

$$G_3 = \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix}$$

Largest clique size: 9
Adjacency Matrix:

$$\begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix}$$

Calculation Time: 82.11 ms
Input Graph 7 :

$$G_7 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

Largest clique size: 6
Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Calculation Time: 72.06 ms

1.4 Maximum common subgraph of two (or more) given multigraphs

In this section we implemented a pretty specific algorithm which works perfectly for ordinary graphs and with some problems for multigraphs. As previously we defined the larges subgraph as a subrgraph with the largest size by our definition of size (number of vertices + number of edges).

1.4.1 Description of the algorithm

1. **Initialization:**
   - Initialize `max_common_subgraph` to `None` and `max_size` to 0. These variables will be used to keep track of the current maximal common subgraph and its size.
2. **Iteration over Subgraphs of the First Graph:**
   - Use `itertools` to iterate over all possible combinations of nodes from the first graph (`all_graphs[0]`). The loop variable `sub_nodes` represents a set of nodes for a potential subgraph.
   - For each combination, create a subgraph (`subgraph1`) using the nodes from the combination.
3. **Checking Isomorphism with Other Graphs:**
   - For each subgraph `subgraph1` created from the first graph, check if it is isomorphic to any subgraph of the same size in the other graphs (`all_graphs[1:]`). This is done using `nx.is_isomorphic`.
   - If there is an isomorphic subgraph in each of the other graphs, set `is_common` to `True`; otherwise, set it to `False`.
4. **Updating Maximal Common Subgraph:**
   - If `is_common` is `True`, calculate the size of the current subgraph (`len(sub_nodes)`).
   - If the size is greater than the current maximum size (`max_size`), update `max_common_subgraph` and `max_size` with the current subgraph and its size.

5. **Return the Maximal Common Subgraph:**
    - After iterating through all combinations, return the `max_common_subgraph`.

1.4.2 Time Complexity Analysis:

Let n be the number of nodes in the largest graph among `all_graphs`, and m be the number of graphs in `all_graphs`.

- The outer loop iterates over all possible combinations of nodes in the first graph, which is $O(2^n)$ in the worst case.
- For each combination, the algorithm checks isomorphism with subgraphs of size k in the other graphs, where k is the size of the current combination. This involves comparing node mappings, which can be $O(n!)$ in the worst case.
- Overall, the time complexity can be expressed as $O(m \cdot 2^n \cdot n!)$, where m is the number of graphs and n is the number of nodes in the largest graph.

1.4.3 Test

The input Graphs :

$$G_1 = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

and

$$G_2 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

and

$$G_3 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Output: Maximal Common Subgraph among all graphs:
Size: 3
Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Calculation Time: 3080.0292 ms
Indeed this matrix describes the triangle which is maximal common subgraph

for the rest of the graphs in file. However , the calculation time is extremely big this algorithm still works precise for the small number of the small graphs.

Chapter 2

Short Technical description

2.1 Language and libraries

The program was written on Python, It used following libraries:
numpy,networkx,itertools,time,scipy, tkinter.

2.2 Description of the GUI

It has a simple user interface with 4 buttons , each one responsible for it's own task. Stage1 - calculates the size of the graph according our definition. Stage2 - distance between the graph. Stage3 - maximum clique in a graph. Stage4 -maximum common subgraph of two (or more) given graphs. Each button will ask you to provide the input file after you pushed it. Also if the program will open in default window you probably will not see the Stage4 button so,please, extend the window and you can see the whole program.
ATTENTION! Please provide the right format of the input file in order to use the application. Otherwise it can cause malfunctioning of the program.

2.3 How to compile the program

The program was tested in Microsoft Visual Code, so all required extensions must be installed before compilation. The Pip functionality for it is required. So the first step is to open folder with our program. To do it firstly you should unpack the archive and in the "File" section choose "Open Folder" option.Then find a folder with our program in source folder "aac_program". Then we must install Pip. How to Install Pip in Visual Studio Code? To install Pip in Visual Studio Code, follow these steps:

-Open Visual Studio Code.
-Click on the "Extensions" icon in the left-hand sidebar.
-Search for "Python" in the Extensions Marketplace search bar.

-Click on the “Install” button next to the “Python” extension.
-Once the installation is complete, restart Visual Studio Code.
-Open a Python file in Visual Studio Code.
-Click on the “Terminal” menu and select “New Terminal”.
-In the terminal window, type the following command and press Enter:
python -m ensurepip –default-pip
-Wait for the installation to complete. Once it’s done, you can start using Pip in Visual Studio Code!
Then with prepared Pip in terminal write the following commands:
pip install numpy
pip install networkx
pip install itertools
pip install scipy
It will install all required libraries to VS code. Then to compile and run the program you can use the terminal command:
python3 program.py
or just open the program.py and push the button ”Run and Debug”.
Of course there is a possibility to just run the .exe file. For it just click 2 times on the ”program.exe” file and you will be able to use the program.

2.4 How to run the program

There are 1 file which could be found in Exe folder: program.exe
In order to run it double click on this file and application will be opened. All necessary instructions on how to use this application you can find in the application.

Chapter 3

Conclusion

In this project we took a look at the problems which can be a part of bigger P=NP problem . We tried to solve it in approximately polynomial time. For some problems it possible to build an algorithm for polynomial approximation of computational time for large graphs. As we found there is exists such algorithm for Maximum clique problem for example which reduce the $O(3^{n/3})$ to approximately polynomial one.
For the isomorphism problem unfortunately there is still no even approximate polynomial algorithm for multigraphs as it is very serious combinatorial problem , but for the purpose of this particular task the VF2 algorithm performs very well.
As for Maximum common subgraph of two (or more) given multigraphs problem , unfortunately, our algorithm performs pretty poor in term of time complexity but we know that the polynomial approximation for this problem can be developed.

References

Hamming distance article on wikipedia

Approximate Maximum Clique Algorithm (AMCA):
A Clever Technique for Solving the Maximum Clique Problem through Near Optimal Algorithm for Minimum Vertex Cover Problem, Muhammad Fayaz, Shakeel Arshad, Abdul Salam Shah and Asadullah Shah

Review of the Bron-Kerbosch algorithm and variations, Alessio Conte

The Graph Isomorphism Problem and approximate categories, Harm Derksen