Implementing Algorithms for matching Biological Graphs: VF2 and VF2 Plus

Advisor: Dr. Carlos Rivero
By: Shraddha Atrawalkar
CONTENTS

1. Abstract
2. Introduction
3. Background
4. Project Goal

5. VF2 Plus Algorithm Overview
6. System Specification

7. Implementation Overview
   7.1 Dataset details
   7.2 File structure
   7.3 Parsing files
   7.4 VF2 Implementation
   7.5 VF2 Plus Implementation

8. Results and Analysis
9. Conclusions
10. Future Work
11. References
1. Abstract

Graphs have been the best form of data structures to represent relationships between entities. Different components of a system, chemical compounds or any real life systems, can be represented as graphs. In the world of bioinformatics, it is extremely convenient to display the molecular representation of a compound in the form of graphs where nodes are elements and edges are the relationship between them. When the need arises to determine whether a compound in anyway is composed of molecules from another compound, a pattern needs to be recognized between the two compounds’ graphical representation. This is the Subgraph isomorphism problem. The task of obtaining matches between a given and target graph is extremely complex and thus, NP-Complete. Over a decade ago, the algorithm VF2 was proposed and is till date considered a reference point in the development of many similar algorithms. But, it is important to note that the size and volume of data represented in graphs was way smaller than what it is today. Thus, in this paper, we will be implementing VF2 Plus, a new and improved algorithm that works on the lines on VF2 with a few additional improvements. The performance of VF2 Plus will be compared to that of VF2 along with the results available for a few more algorithms like RI, LAD and L2G for the same set of graphs.

2. Introduction

In the recent times, there has been a surge in the amount of data available to us. But, this data holds no importance unless represented in a well understood form. A large amount of this data is relational and what better structure than a graph to represent this kind of data. Graph is a well suited data structure to represent information from varied sources including network topology, social circles, chemical components, city maps etc. Graphs have already been proved of great importance during social network analysis where each node in a graph represents a person and edges in the graph represent the relationship between 2 people or just the fact that they are connected to one another. Similarly, we can represent chemical compounds as graphs, where each node represents a molecule in the compound and edges represent the interaction between the molecules. So now, when we start analyzing these graphs, there are often times where we need to find common patterns between 2 graphs i.e. we need to detect whether 2 given graphs are similar to each other. This is the isomorphism problem. Let us consider the following graphs to get a better understanding of the problem.

![Figure 1 Graph Isomorphism Example [1]](image-url)
Let’s call the 2 graphs A and B (from L-R). Say, we want to find whether graph A is isomorphic to graph B. When we represent our problem mathematically, we can say we need \( f: V(A) \rightarrow V(B) \). We need a function where we can map vertices from graph A to vertices of graph B. If we have such a function between 2 graphs, we can say that one graph is isomorphic to the other. From fig. 1 we can see that, there does exist a function, where \( a \rightarrow 1, c \rightarrow 2, e \rightarrow 3, b \rightarrow 4 \) and \( d \rightarrow 5 \). Now consider fig 2, here we want to find whether G1 is a subgraph of G2. Not only this, we also want find out if the structure of the subgraph in G2 is the same as that of G1.

![Figure 2 Subgraph Isomorphism Example [2]](image)

Here, although G1 has less number of nodes as compared to G2, we can obtain a function that maps few nodes from G2 \( \rightarrow \) G1. It is important to note that, when identifying whether a graph is an isomorphic subgraph of another graph, we need to preserve the structure of nodes in the graph. Thus, say we consider G1 and map it to \((10\rightarrow 7\rightarrow 3\rightarrow 4\rightarrow 8)\) from G2. Thus, we can say that G2 has a subgraph that is isomorphic to G1. But, detecting subgraph isomorphism is not as straightforward as it seems. The problem grows more complex as the number of nodes in the graph keeps increasing. We cannot keep comparing each node to check its neighbors to look for a corresponding match in the target graph, certainly not in polynomial time. Subgraph Isomorphism problem is known to be an NP-complete algorithm. Thus, we need an algorithm that reduces the computation time and uses a graph’s structural information to solve the problem. VF2 is one algorithm that has helped solve graph isomorphism problems for over a decade. But, as the challenges in the fields where graphs are used have increased, we need an improvement over VF2 and we call it VF2 Plus. VF2 plus helps solve the subgraph isomorphism problem in biochemical graphs by implementing a few improvements over the original VF2 algorithm.

3. Background

As we know, the subgraph isomorphism problem is a very well-known problem and many algorithms have been proposed to solve it. The most straightforward technique to solve this problem is to enumerate all possible subgraphs of the target graph and then compare them to our given graph. It is of course obvious that this wouldn’t be the best approach when the size of
graphs grows considerably large. One of the well-known algorithms proposed is by Ullmann[3]. His algorithm implements a backtracking approach along with Depth First Search. This algorithm implemented a strategy to check all the nodes and then decide whether they would be a part of the solution set. Another strategy implemented by McGregor[4], aims to exclude nodes in a graph that would definitely not be a part of the solution. Lastly, another strategy implemented by algorithms like GraphQL[5] suggest only determining whether target graph contains a given graph. It does not determine the particular subgraph that can be mapped to the given graph. Despite of having the above algorithms to solve our problem, we still need an algorithm that not only implements positives of the above algorithms but is also flexible enough to adapt to graphs implemented in different areas of study and solve the same problems.

This algorithm is VF2, which does not take into consideration the physical properties of the graph, and thus makes it flexible in solving subgraph isomorphism problem in various types of graphs. This algorithm is implemented using a set of feasibility rules. These rules help reduce the search space of the algorithm, making the process of matching much faster. The algorithm divides the implementation into various states and at each state forms a search space and partial mapping set. It maintains that if a pair of nodes is in the mapping set in at least one state, it is surely a part of the final mapping set. The algorithm also implements a verification step, where it adds one of the neighboring nodes in the mapping set. If this addition violates the subgraph isomorphism rule, the entire path from the mapping node is discarded, thus saving computation time down the path from that node.

The algorithm starts by computing sets with nodes at each state namely:

- **Core set (M(s))**: This set contains the pairs of mapped nodes
- **Terminal sets (T(s))**: It computes 2 such sets. They contain nodes which are not yet part of the core set. (T1(s) and T2(s)).

The number of terminal sets increases in case of directed graphs. T1^{in}(s) and T2^{in}(s) and T1^{out}(s) and T2^{out}(s).

Before proceeding to the next state, it selects a candidate pair (u,v) where u ∈ T1(s) and v ∈ T2(s).

Now, the algorithm implements a strategy to avoid computation for a pair of nodes that would for sure not be a part of the mapped set. It implements 5 feasibility rules to detect such nodes early on:

- **0 Look-Ahead**:
  a. R_pred: if in the partial mapping set, every predecessor of u corresponds to every predecessor of v and vice-versa.
  b. R_succ: if in the partial mapping set, every successor of u corresponds to every successor of v and vice-versa.

- **1 Look-Ahead**:
  c. R_termin: if the number of predecessors /successors of u in T2^{in}(s) is greater than or equal to that of v in T1^{in}(s).
  d. R_termout: if the number of predecessors /successors of u in T2^{out}(s) is greater than or equal to that of v in T1^{out}(s).

- **2 Look-Ahead**:
  e. R_new: if the number of predecessors /successors of u that are neither in M2(s) or T2(s) are greater than or equal to that of v in M1(s) and T1(s).
If a pair \((u,v)\) fails to satisfy these feasibility rules, no further computation is performed down the path of these nodes.

Also, even though VF2 is considered to be one of the best algorithms for subgraph isomorphism problem, it has certain weak points. In this paper, we’re trying to implement the algorithm VF2 Plus which helps overcome weaknesses of VF2. We will mainly be talking about two main weaknesses: total order relationship and structure of the terminal sets.

As a part of our problem, we are dealing with biological graphs mainly proteins. We need an algorithm to match graphs of 2 proteins and identify whether one is subgraph isomorphic to another. Protein structures are extremely complex and a minor change in their structure completely changes the protein and its properties. We will be implementing VF2 plus algorithm and testing its effectiveness on the RCSB database [6].

4. Project Goal

In this project, we want to implement the VF2 plus algorithm which is an improvement on the VF2 algorithm. We want to test our implementation on the RCSB dataset and compare its efficiency to the VF2 algorithm. For this, we need to first get the dataset, parse its files and generate graphs from the dataset. Once we have the graphs ready, we will test our implementation and finally report our findings.

5. VF2 Plus Algorithm Overview

Since we’ve already discussed VF2 in detail and have a fair understanding, we can now discuss the improvements we wish to implement as a part of VF2 plus. As seen earlier, VF2 requires a search space to choose the next candidate pair. This means, even if a node is not going to be a part of the mapping solution, the algorithm will have to spend some time deciding this. Also, it has to start at the start node and check every node in the path to find a match.

Instead of this, if it is pre-computed which nodes are (probably) going to be in the solution and provided to the algorithm, it will make the mapping process faster. Now, to pre-compute, we will calculate the probability of each node depending on its connections in the graph and generate a list of nodes. This list will be provided to the graph, and now before moving to the next state, the algorithm doesn’t spend time looking for the next candidate pair, thus reducing its search space. It can simply look at the next node in the list and continue with its matching process.

In order to calculate the probability of the node, the algorithm considers the structural as well as semantic properties of the graph i.e. labels on each node and edges connected to each node. Once we have the probabilities, we need to sort the nodes. For this, we follow the steps as below:

Let \(M\) be a set of sorted nodes, \(T\) be a set of nodes to be sorted, \(\text{degree}_M\) be number of edges connected to a node in \(G_1\).

a. We will select the node which has the lowest probability. In case, there is more than one node with the same lowest probability, we will select the node with the highest degree. If we again have more nodes with the maximum degree, we will select the first node.

b. Since we have a node selected, we will add it to set \(M\), and nodes connected to this node will be added to set \(T\).

c. Now, from set \(T\), we will select the node with the highest \(\text{degree}_M\). Again, if we have more than one node with the highest \(\text{degree}_M\), we will select the node with the lowest probability.
If we have more nodes with the same lowest probability, we will select the first node. Once we select the node, we will repeat b.

From VF2 we know that each time the algorithm wants to find the next candidate pair for the next state, it has to look at the terminal sets. Thus, we know that the number of nodes in the terminal sets highly affects the computation complexity of the algorithm. If we can reduce the search space of the algorithm by reducing the nodes in the larger graph, we may be able to improve the efficiency. For this, we analyze the neighbors of the already mapped node v from the larger graph to that of the smaller graph u. Also, the algorithm implements a classification of the terminal set. For this, we generate smaller subsets of the terminal set and apply feasibility rules on each of them. Like this, we can considerably reduce the number of possible candidate pairs.

6. System Specification

Operating System: Windows 8.1
Coding language: Java
Libraries: JGRAPHT, JGRAPHX
Environment: Eclipse Neon

7. Implementation Overview

We begin our implementation by retrieving our dataset from the ICPR website. Once on the website, we need to create an account in order to access the dataset and details related to it. Under the ‘Datasets and results’ tab, we can find the complete dataset.

Dataset details:

We have 3 graph datasets: Molecules, Contact Maps and Proteins. Each of these datasets contains 3 sub-folders namely: query, ground_truth and target.

The folders query and target contain graph files with a .grf extension. The target files are named as [biological_structure].grf, whereas, the query files are named as [biological_structure].[query_size].sub.grf and the ground_truth files are named as [datasetname].[query_size].gtr. A few constraints have been imposed on the graphs like, all nodes are named with strings up to 5 characters, edges are directed and not labeled, no 2 nodes have more than one edge between them and no cycles exist in the graph.

File structure:

The graph files in **query and target** are formatted as follows:

<table>
<thead>
<tr>
<th>Line detail</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Number of nodes in the graph</td>
</tr>
<tr>
<td>0 C</td>
<td>Label for each node</td>
</tr>
<tr>
<td>1 C</td>
<td>Number of outgoing edges from the node described in the next lines</td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Details of outgoing edges from the node.
(here, edge from 0 to 1 and 0 to 2)

The file structure of **ground_truth** files is as follows:
- **T**: [Target Graph File]
- **P**: [Pattern Graph File]
- **N**: [Number of Solutions]
  - [Solutions details]
  - The solution details are given in a single line for each solution.
- **S**: [No. of Solutions]:[Mapped Pairs]

The following is a snippet from the ground_truth file:
- **T**: backbones_1O54.grf
- **P**: backbones_1EMA.8.sub.grf
- **N**: 32
- **S**: [8]: [0,333;1,334;2,337;3,338;4,339;5,340;6,341;7,347]

**Parsing files:**

We now need to read these files and generate graphs for each of the file. For this, I have written a Java program. The program takes as input a .grf file and starts reading file line by line. It gets the number of node in graph and creates an adjacency matrix to store node details later. It reads the next set of lines and generates a map of [Node number: Label]. It then reads the number of edges to be described in the next lines. It then reads the edge details and creates an entry in the adjacency matrix for the edge. Since this is a directional graph, we will only have an entry for \((node \ u \rightarrow node \ v)\) and never \((node \ v \rightarrow node \ u)\) or vice versa. This is because as mentioned earlier, only one edge exists between 2 nodes and there are no cycles in the matrix.

To actually plot graphs, we will be using the JGRAPHT library. We simply loop through the adjacency matrix to add edges to the directed graph and through the map to add vertices/nodes to the graph.

![Graph generated from a sample .grf file](image)

We then pass this directed graph into a mxGraphComponent which is provided by the JGRAPHX library. We need to provide our directed graph’s nodes and edges and the component plots it on to the screen.
**VF2 Implementation:**

The VF2 algorithm takes as input a state $s$ and returns the mapping i.e. matching pairs between the two graphs considering the current state only. The implementation of this algorithm works in states and the main goal is to first eliminate all those nodes that will not be a part of the final mapping.

VFS algorithm takes a state $s$ as input. It considers terminal sets $T(s)$ for each graph and mapping set $M(s)$ for the pair of graphs being considered. It computes a set of candidate pairs $P(s)$ that could potentially be a part of $M(s)$.

**Pseudocode:**

VFS($s$):
1. IF $M(s)$ contains all the nodes in the graph, return $M(s)$
2. ELSE
   WHILE $T(s)$ is not empty:
      a. create set $P(s)$ with candidate pairs to be included in $M(s)$, this will be all the nodes not yet considered for mapping
      b. FOR every pair $p$ in $P(s)$
         Apply feasibility rule to each pair,
         IF, $p$ satisfies $R_{\text{pred}}$ && $R_{\text{succ}}$ && $R_{\text{termin}}$ && $R_{\text{termout}}$ && $R_{\text{new}}$:
            Add $p$ to $M(s)$ and create a new state $s'$
   VFS($s'$)

As we begin the implementation, our first state is $s_0$. At this point, since we haven’t yet started our mapping process, $M(s_0)$ will be empty. Each state will contain a pair of nodes that are a part of the mapping set. This means that at the end of processing the current state, we will either exclude or include the current pair in consideration, a part of the final mapping.

![Figure 4 Example graph for VF2 [7]](image)

Consider Fig 4, let us start with nodes (n1, m4):

- If mapping set covers all nodes: At this point, if both the graphs contain only nodes n1 and m4, our search for mapping nodes is complete and we return (n1, m4) as the mapping set.
- Else, we choose candidate pairs from the terminal set, which is the set of nodes that haven’t been considered yet. Say, we choose (n2, m1) as the next pair of nodes. We need to check if adding this pair, will lead us to our goal state (which is a set of mapped nodes). For this, we will apply feasibility rules on this pair, and include exclude them depending on whether the feasibility rules succeed on this pair.
• Say, feasibility rules succeed on this pair, we will add the pair to the mapping set and call the algorithm again passing the mapped set.
• If the feasibility rules do not succeed on the pair, we know that if a pair isn’t a part of the partial mapping set, it will not be a part of our goal mapping set. Thus, we will discard this pair and look for another candidate pair and continue the process, till we reach the end of one of the graphs.
• Our goal is to obtain a partial mapping set at the end of each state.

Since we have been talking about feasibility rules so much, let see in detail, what feasibility rules are. There are 5 look ahead rules that we are going to focus on for VF2. Let us consider the partial mapping set \( M(s) = \{(n_1, m_4), (n_2, m_1)\} \). The look ahead rules are implemented as follows:

A. **0-Look Ahead:** In these rules, we only look at nodes that are in a child parent relationship.
   1. **R_pred:** We say that a particular pair of nodes satisfies this rule iff for a pair of nodes \((n, m)\), there exists a predecessor \(n'\) for node \(n\) and \(m'\) for node \(m\).
      As per our example, we can say that \((n_2, m_1)\) satisfy \( \text{R}_\text{pred} \) feasibility rules since in the partial mapping set, \(n_2\) has its predecessor \(n_1\) and \(m_1\) has its predecessor \(m_4\).
   2. **R_succ:** We say that a particular pair of nodes satisfies this rule iff for a pair of nodes \((n, m)\), there exists a successor \(n'\) for node \(n\) and \(m'\) for node \(m\).
      As per our example, we can say that \((n_1, m_4)\) satisfy \( \text{R}_\text{pred} \) feasibility rules since in the partial mapping set, \(n_1\) has its successor \(n_2\) and \(m_4\) has its successor \(m_1\).

B. **1-Look Ahead:** In these rules, we look at the number of nodes in the terminal sets of both the graphs
   3. **R_termin:** We say that a particular pair of nodes satisfies this rule iff for a pair of nodes \((n, m)\) the number of predecessor nodes for node \(n\) in \(T_1^{\text{in}}(s)\) is greater than or equal to the number of predecessor nodes for node \(m\) in \(T_2^{\text{in}}(s)\) and the number of successor nodes for node \(n\) in \(T_1^{\text{out}}(s)\) is greater than or equal to the number of successor nodes for node \(m\) in \(T_2^{\text{out}}(s)\).
      As per our example, we can say that \((n_2, m_1)\) satisfy this rule as currently both nodes have equal predecessor nodes which is 0 and 2 successor nodes each.
   4. **R_termout:** We say that a particular pair of nodes satisfies this rule iff for a pair of nodes \((n, m)\) the number of predecessor nodes for node \(n\) in \(T_1^{\text{out}}(s)\) is greater than or equal to the number of predecessor nodes for node \(m\) in \(T_2^{\text{in}}(s)\) and the number of successor nodes for node \(n\) in \(T_1^{\text{out}}(s)\) is greater than or equal to the number of successor nodes for node \(m\) in \(T_2^{\text{out}}(s)\).
      As per our example, we can say that \((n_2, m_1)\) satisfy this rule as currently both nodes have equal predecessor nodes which is 0 and 2 successor nodes each.

C. **2-Look Ahead:**
   5. **R_new:** We say that a particular pair of nodes satisfies this rule iff for a pair \((n, m)\) the number of predecessors and successors of \(n\) neither in the mapping set \(M(s)\) and terminal set \(T(s)\) is greater than or equal to that of \(m\).
      As per our example,
\[ \{n1, n2\} \quad \{n3, n4\} \]
\[ \{m4, m1\} \quad \{m2, m3\} \]

If we want to check the rules for \((n2,m1)\), we can see that,

- Number of predecessors of \(n2\) not in \(M1(s)\) = 0
- Number of predecessors of \(m1\) not in \(M2(s)\) = 0
- Number of successors of \(n2\) not in \(T1(s)\) = 0
- Number of successors of \(m1\) not in \(T2(s)\) = 0

Since \((n2, m1)\) satisfies all the above conditions, it satisfies this rule.

Therefore, we can see how to verify and discard node pairs based on whether a particular node pair satisfies all feasibility rules.

We can now see why it is important to discard nodes rather than find a match early in the computation. Eliminating nodes reduces the search space thus speeding up the process of finding matching nodes.

**VF2 Plus Implementation:**

The VF2 Plus algorithm is an improvement over the VF2 Algorithm for subgraph matching. The VF2 Plus algorithm, tries to reduce the search space even before the matching process begins. This affects the computation time considerably and VF2 Plus is thus able to process the graphs and get matching nodes much faster and efficiently as compared to VF2.

Let’s take a look at what improvements need to be implemented to reduce the search space for VF2 Plus:

Order nodes based on probability: VF2 Plus computes a probability of each node being a part of the solution set. It makes use of the structural information provided by the graphs to compute the probabilities. In order to implement this feature, we first compute the frequency of each label in each of the graphs. After this, we compute the degree of each node in each graph. We then make use of the following formula and calculate the probability of each node in the query graph making use of the degree and label frequencies for the target graphs. The formula is given as

\[ P(u) = Plab(L) \times d' \geq d \times Pdeg(d) \]

So here, we compute the probability for every node in the query graph ‘\(u\)’. For this, we check if there exists the same label in the target graph. If the label does not exist, the probability is set to 0. If it does exist, we check for the degree of that node, the degree of the node is a sum of both inbound and outbound edges to the node. The number of inbound and outbound edges can be obtained from the ‘DirectedGraph’ object’s ‘inDegreeOf’ and ‘outDegreeOf’ methods. Each of these methods takes as input the node object and return an integer which the indegree or outdegree of the node. Once, we have the degree and the frequency of nodes, we compute a product of these 2 values which then each divided by the total number of nodes in the graph. Thus, giving us the probability of each node in the query graph. We store these values in a map, where the keys are probabilities and the values are a list of nodes with that frequency. The map is sorted in ascending order based on the probability values.

We now need to order the search space in such a way that the nodes which have a probability of being the solution set are computed ahead of the others.

The following is a pseudo code for computing the ordered set of node.
Note that the degreeMap consists of the key as degree and value is a list of nodes with that degree and probabilityList is sorted in ascending order

Pseudocode:

orderNodes(queryGraph, probabilityList, degreeMap):

1. We first select the node with the lowest probability.
   a. If number(lowestProbabilityNodes) > 1, getMaxDegreeNode from degreeMap
   b. If number(maxDegreeNodes) > 1, select 1st node from list.
2. Add node to M(s) and its neighbours to T(s)
3. Select node with maxDegree from T(s)
   a. If number(nodeWithMaxDegree) > 1, getLowestProbabilityNode from probabilityList
   b. If number(lowestProbabilityNodes) > 1, select 1st node from list.
4. Repeat 2 till no terminal nodes.

8. Results and Analysis

Fig 3 shows a sample graph plotted using a sample .grf file. As we can see, the graph is directed, nodes are labeled and there is no cycle in the graph. Also, no two nodes are connected by more than one edge between them.

Once we are able to parse our files and generate graphs, we can then implement this code to read and generate graphs from all files provided as target and query graphs. After that, we can start implementing our VF2 plus algorithm and provide our graphs as input and analyze how efficiently it is able to detect whether a graph from query folder is subgraph isomorphic to a graph in the target folder.

As seen in section 7.4, VF2 algorithm, after each iteration provides us with a state. This state is a set of nodes that forms the partial mapping set. We also update the mapping set M(s) and terminal set of each graph T(s) after each iteration and discard/include nodes thus reducing our set of candidate nodes from each graph. Each node in this partial mapping set is guaranteed to be a part of our goal set which is the set of nodes in the final mapping set since they have been put through a test of feasibility rules.

Implementing VF2 Results:

Consider the following inputs and mapping output produced:

*We shall consider A->0, B->1, C->2 and so on…*

The output after executing the subgraph matching algorithm, we get:

T: a.grf
Here, 0->A, 1->B and 2->C is the mapping of nodes from (a) -> (b). (a) is exactly contained inside (b) i.e. the structure of the graph, its labels and edges (A-B and A-C) exactly match to those in (b) thus, we can say (a) is isomorphic to (b).

Thus, we get S as the set which returns which node from (a) matches with which node from (b).

Consider another pair of graphs as below:

We have (b) which has larger number of nodes than (a). Thus, we need to find a pattern (a) inside (b) and only then can we say that (a) is isomorphic to (b).

The output after executing the subgraph matching algorithm, we get:
T: a.grf
P: b.sub.grf
N: 1
S: 3: 0,1;1,3;2,2
Thus, we can map from (a)->(b) nodes. Edges (B-A, C-B & C-A) from (b) map the entire graph (a) within it.

Remember, we need to preserve the structural information from the nodes i.e. along with the nodes that a node is connected to, the labels of the nodes should also match in both the graphs. Only then can we say that we have performed an exact matching.

VF2 Plus Implementation Results:
Considering the same graphs as above, let’s see how VF2 Plus performs:
We first compute the frequency of each label and degree of each node in the target graph:

<table>
<thead>
<tr>
<th>Label</th>
<th>Frequency</th>
<th>Node</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

Now, we will calculate the probability of each node in query graph using the formula mentioned above:
e.g. for node A \( P(A) = (1) \times (3/4) = 0.75 \)

Similarly, calculate probabilities of other nodes as B(0.5), C(0.5) and D(0.25)

Then, we compute the mapping and terminal set using the VF2 Plus ordering procedure and get ordered set as (1,2,3,4);

Now similar to VF2 we apply feasibility rules to each pair of target graph nodes and ordered set and obtain the mapping result as (0,1;1,3;2,2)

**Comparing execution times for VF2 and VF2 Plus:**

The dataset contains graphs of 3 types: Molecules, contactMaps and Proteins. In all, there are 10601 target graphs and 170 query graphs. These graphs range from 4 to 256 nodes and are all directional, acyclic and can have multiple nodes in a single graph with the same label.

The implementation of algorithms was tested for all query and target graphs and a comparison of their average execution times is as follows:

From the above graph, we can see that both the algorithms work fairly similarly in terms of time taken for execution for smaller graphs, but as the number of nodes increases, VF2 takes an exponentially large amount of time as compared to VF2 Plus.

This is because, VF2 Plus using the ordering of nodes and eliminating them before starting the process of matching, reduces the search space considerably. Whereas, VF2 applies a recursive DFS approach visiting each node at least once, and in fact more number of times during the executing, thus taking a longer time to compute the solution.

The results from both the algorithms were compared to those given in the ground\_truth files for pairs of query and target graphs and the accuracy was 98.46% and 98.31% for VF2 and VF2 Plus respectively. This is because; there is an inconsistency within the ground\_truth results where for some pairs of graphs, the results are computed for an exact match, whereas for others they are inexact matches. The implementation for VF2 and VF2 Plus in this project implements an exact match where the query graph is exactly matched to the target graph.
9. Conclusions

As a part of this project, we implemented algorithms VF2 and VF2 Plus which is an improvement over the former. From the results it can be concluded that, although VF2 is among the best algorithms there is for graph matching, VF2 Plus performs even better for larger graphs. The improvements we made on VF2 i.e. ordering nodes and classifying terminal sets reduces the search space before the mapping process starts and considerably helps VF2 Plus in computing results much faster. When we use the structural information of the nodes to compute the probability of nodes and sort the nodes, we restrict the search space and discard the graphs that do not match very early on in the matching process. Also, building subsets of terminal sets and applying feasibility rules to these subsets, helps us discard those candidate pairs early on. Because of this, we don’t even consider these candidate sets as potential pairs while computing the matches. Both of these improvements have helped VF2 Plus, traverse through the graphs and take much lesser time to get the matching pairs of nodes.

Thus we see that both algorithms start off by performing similar to each other for smaller graphs, but as the number of nodes increases (here, 32) the time taken by VF2 increases exponentially whereas VF2 Plus continues to perform better. VF2 takes the recursive DFS approach where it considers every node in the graph more than once (in case of multiple matches) thus it repeats a lot of subgraph matching taking a much longer time. All this is skipped by VF2 Plus as it knows exactly which nodes to consider and which to ignore taking a much faster route. Therefore, if we have a problem where we have a large number of nodes in the target or query graph, we should use VF2 Plus graph matching algorithm and for smaller graphs we should use VF2 algorithm.

10. Future Work

The dataset considered for this project consisted of graph files with number of nodes in the range of 4-256. In order to parse the graph files, generate graph objects and test the implementations on larger than 256 node graphs, we need to devise a way to generate graph files for these larger graphs. The graph files have a specific structure and in future, we will need to generate graph files of the similar structure to test our implementation on larger graphs. Also, VF2 Plus requires generating subsets of candidate nodes and applying feasibility rules to these subsets in order to eliminate them. But, this entire process itself is very taxing. We need to improvise on the technique in which we are eliminating subsets of nodes using classification techniques which are computationally less expensive.

11. References

[2] Vincenzo Carletti, Pasquale Foggia, and Mario Vento, ‘VF2 Plus: An Improved version of VF2 for Biological Graphs’, University of Salerno, Salerno, Italy
