Implementation and Analysis of Efficient Subgraph Matching Algorithms in Memory

Submitted By
Madhusudan Hanagal
May 2017

Advisor
Professor. Carlos R.Rivero
Rochester Institute of Technology
10. Conclusion
11. Future Work
12. References
i. Figures

<table>
<thead>
<tr>
<th>Figures:</th>
<th>Page No:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1: Framework created by Prof. Carlos R. Rivero</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2: VF2[1] Algorithm working</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3: Adding candidate pair to state[1]</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4: GraphQL Algorithm[2]</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5: Example Graph G and Pattern graph G</td>
<td>14</td>
</tr>
<tr>
<td>Figure 6: Refine Search Space</td>
<td>17</td>
</tr>
<tr>
<td>Figure 7: Common Example</td>
<td>18</td>
</tr>
<tr>
<td>Figure 8: Dataset Structure</td>
<td>21</td>
</tr>
<tr>
<td>Figure 9: Project Structure</td>
<td>22</td>
</tr>
<tr>
<td>Figure 10: Maven pom.xml</td>
<td>24</td>
</tr>
<tr>
<td>Figure 11: Sample output</td>
<td>25</td>
</tr>
<tr>
<td>Figure 12: VF2 vs GraphQL nodes</td>
<td>26</td>
</tr>
<tr>
<td>Figure 13: GraphQL query size</td>
<td>26</td>
</tr>
<tr>
<td>Figure 14: VF2 vs GraphQL datasets</td>
<td>27</td>
</tr>
</tbody>
</table>
ii. Tables / Algorithms

<table>
<thead>
<tr>
<th>Tables:</th>
<th>Page No:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: Inverse adjacency set creation</td>
<td>14</td>
</tr>
<tr>
<td>Table 2: Finding feasible mates</td>
<td>14</td>
</tr>
<tr>
<td>Table 3: Search Order</td>
<td>16</td>
</tr>
<tr>
<td>Table 4: VF2 Working</td>
<td>18</td>
</tr>
<tr>
<td>Table 5: Search Space Phi</td>
<td>19</td>
</tr>
<tr>
<td>Table 6: Build Query map and Graph map</td>
<td>19</td>
</tr>
<tr>
<td>Table 7: After NPP</td>
<td>20</td>
</tr>
<tr>
<td>Table 8: Default search order</td>
<td>20</td>
</tr>
<tr>
<td>Table 9: Final matched output</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithms:</th>
<th>Page No:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1: VF2 Algorithm[1]</td>
<td>9</td>
</tr>
<tr>
<td>Algorithm 2: Phase 1 Graph Pattern Matching</td>
<td>13</td>
</tr>
<tr>
<td>Algorithm 3: Phase 2 Refine the search space[2]</td>
<td>14</td>
</tr>
</tbody>
</table>


1. Abstract

Subgraph isomorphism, alternatively known as subgraph matching is a concept using which we can find all the isomorphisms i.e. given a datagraph and a pattern (query) graph, we find all matching of pattern graph that are present in data graph. As subgraph matching has various applications in bioinformatics, DNA sequencing and so on. For this, it is required that we find the best subgraph matching algorithm whose performance is better and efficient when dealing with larger data graphs. This project implements and compares two such algorithms called as VF2[1] and GRAPHQL[2]. VF2 has two isomorphisms, graph-graph isomorphism and graph-subgraph isomorphism. For this project we implement the VF2(Graph- Subgraph)[1] Isomorphism, and for GRAPHQL[2] we implement the graph-query(subgraph) isomorphism. This report covers detailed explanation, working and an example each for both the algorithms. Lastly, the report shows the results obtained by implementing the algorithms on Proteins datasets. A comparative analysis shows which algorithm to choose in different scenarios. This paper also contains a file parser implementation to suit the given datasets[3] and their structure.
2. Introduction

Data that resides in many domains can be modeled using graphs. These models can be functional in different fields like Bioinformatics, Chemoinformatics and so on. A protein can be represented as a set of remains [2] where remains represent the nodes and edge represent the spatial proximity [2]. Atoms and chemical bonds can also be modeled as graphs in Chemoinformatics [2]. With the increase of the data models and size of data models, it is very hard to maintain, query and manage these data models [2]. To achieve a high performance query processing system we need to develop sophisticated graph querying algorithms. Considering GRAPHQL [2], some of the examples that can be related to different domains are as follows:

- Given an aromatic ring as query find all heterocyclic chemical compounds [2]
- Alpha and beta barrel in protein compounds [2].
- RDF: find all instances of Resource Description framework [2].
- Finding co-authors from from set of papers (DBLP) [2].

The important thing about GRAPHQL is that all the graphs which are taken as input and queried, should produce a collection of graphs [2]. Pattern recognition is a wide area of study in computer science, which considers graphs closely [1]; a pattern in a graph is nothing but the subgraph matching isomorphism. The two types of matching that concerns the subgraph isomorphism, they are:

- Exact Matching: Find all the details that makes two graphs isomorphic. It is strict matching [1]
- Inexact Matching: Find the similarity, but not the exact details [1].

VF2 is the exact matching technique, which is very helpful in applications in pattern recognition domain [1]. The biggest concern of subgraph matching is that it is also NP-complete. Many algorithms have been developed like Ullman’s subgraph matching algorithm which takes O (n^3) time, but in this case the graphs are smaller in size [1]. Ullman’s algorithm also deals with both graph-graph isomorphism and graph-subgraph isomorphism. VF2 is a deterministic approach that has a state space representation of the graph. It is considered as State, and five feasibility rules for pruning the search tree [1]. This main problem that has been tackled in this project, is to implement and analyze the working of these two algorithms and then integrate with the framework. The framework developed by Professor Carlos Rivero implements and studies different algorithms, which are analyzed in the framework. The framework has to integrate with VF2, which is the only exact matching that is available. This report also shows the framework developed by Professor Carlos Rivero in the architecture part of the report. Keeping this idea in mind we were able to develop the VF2 algorithm and as the future work to this project, the algorithm can be implemented with the framework. A brief section on the future work and scope is discussed in the latter part of the report. This report does not implement the Ullman’s algorithm but it does show analysis by comparing VF2 and GRAPHQL algorithms with Ullman’s algorithm. The related paper of VF2 [1] generates some graphs and attributed graphs on which the authors test the algorithm. In order to see how fast each algorithm works, Proteins dataset is used for both and is checked against the ground truth.
3. Background / Motivation

The authors of VF2[1] have published 3 papers on the algorithm which are VF[4], VF2[1], VF2-Plus[5]. Also they have published a paper on performance evaluation of the VF[4] algorithm. The VF2 algorithm addresses search space problem in VF [4] algorithm, by organizing the different data structures[1], making this algorithm more efficient when matching larger sized graphs(which contain more than thousand nodes). Authors of GRAPHQL[2] mainly focused on building a kind of query language which might be useful in querying the graph systems[2]. They took into consideration the traditional database system for example RDBMS, which uses different operators to obtain the results, similarly the GRAPHQL algorithm also makes use of selection operator, kleene star operation, concatenation and disjunction operators to get the results.

Professor Carlos Rivero has built a framework shown in the below figure which consists of different subgraph matching algorithms, this framework helps the user to chose any algorithm and then analyze the results by running these algorithm on different kinds of datasets. The main reason of choosing this project was to build VF2 algorithm which is not in the framework and integrate it. This algorithm was built as an in memory algorithm. Also analyzing the performance of VF2 by comparing with GRAPHQL. The highlighted portion of the framework is where the algorithm VF2 fits in.
4. Algorithms

4.1 VF2

4.1.1 Overview

The VF2[1] algorithm is a subgraph matching algorithm which finds all the feasible mappings between two given graphs. This algorithm is Exact matching algorithm. The algorithm has two different kinds of matching(isomorphism), a graph-graph isomorphism and a graph-subgraph isomorphism. It is a 2nd version of the algorithm VF[4] and mainly handles the search space problems that were encountered in version 1 of algorithm.

4.1.2 Definition

Input: $G_1(N_1, B_1)$ and $G_2(N_2, B_2)$ [1]
Output: all possible mapping between a node in $G_1$ and a node in $G_2$ and vice-versa[1].
Constraints: 5 feasibility rules are defined to find the mappings. A state space representation (state) which has two types $s_0$ and an intermediate state [1].

4.1.3 Implementation

Given two graphs $G_1$ and $G_2$, a initial state $s_0$ which initializes all the data structures is obtained. Then an instance of this algorithm is passed on to the matcher method which checks if there exists any feasibility between the two pairs of nodes taken from $G_1$ and $G_2$. If the feasibility check return true then the candidate pair is added to the state which makes a partial matching and then the routine is continued for the next candidate pair till all the pairs are exhausted. If the match is found then we backtrack to the last point of candidate pairs else we return if one matching is found. In order to find all mappings we need to print each state that has the pair which has passed the feasibility check. The following snippet shows the pseudocode for the algorithm[1] and then follows the explanation of feasibility check rules.

Mapping = $\{(Q, P) \in N_1, N_2 | Q$ is mapped onto $P \}$[1]

shows the mapping equation.
Input: (datagraph,querygraph) where sizeof(querygraph) < sizeof(datagraph) in case of Graph-subgraph isomorphism.
    or sizeof(querygraph) = sizeof(datagraph) in case of graph-graph isomorphism.
Output: List<mappings> and N<total number of matches>
    A set of mappings and number of matches
Algorithm:
    Step1: Initialize the state (s₀)
    Step2: call Match(s₀)
    Step3: Match(s₀)
        if depth of search tree is reached
            print the mapping
        else
            for: set of candidate paris
                check(feasibility(node1,node2))
                Call Match(s₀)
                // We can backtrack after the mapping is found
                // to get all mappings
                if(!found)
                    backtrack();
                else
                    return true; // returns only one mapping
    Step4: check(feasibility(node1,node2))
        Predecessor rule()
        Successor rule()
        In node rule()
        Out node rule()
        New nodes rule()

Algorithm 1: VF2 Algorithm[1]

There are 5 feasibility rules that each candidate pair should adhere to in order to enter the matched set of paris. The following points explain all the 5 feasibility rules, but before that we need to know some of the important data structures and variables that are maintained to perform the feasibility operations, here are the variables T_{out}^x and T_{in}^y were in and out denotes the incoming and outgoing edges and x and y represent (1,2,....n).

- Predecessor rule: if there exists a node a and it is suppose a predecessor of node b then there exists c which is predecessor of d [4].
- Successor rule: if there exists a node a and it is suppose a successor of node b then there exists c which is successor of d [4].
- Termin rule: predecessor(T1 in ) is equal to successors(T2 in) for a given node [4].
- Termout rule: predecessor(T1 out) is equal to successors(T2 out) for a given node [4].
- New node rule: pred(m1 not in T1) != pred(m2 and T2) [4].
4.1.4 Example

The simple example below shows the working of VF2 algorithm:

As we can see from the above figure that same color nodes show the mappings of the two graphs. First the initial state $s_0$ is initialized. $\text{core}_1[1]$ and $\text{core}_2[1]$ hold the actual mappings and are equal to size of $\text{nodes}[1]$. $G1$ and $G2$. In the above example $\text{core1}$ is set to 4 and $\text{core2}$ set to 4, $\text{core1}$ of size $n$ contains $\text{index}[1]$, $\text{in}_1$, $\text{in}_2$, $\text{out}_1$, $\text{out}_2$ whose dimensions include the size of the corresponding graphs. The initial state is passed to the match function which takes the first two candidate pairs ($A$ and 1) then it checks the feasibility rules for these candidate pairs, $A$ has one outgoing edge (Termout rule) and one incoming edge (Termin rule) in graph 1 and node 1 has two outgoing edges. So this is not an ideal pair as it fails the check. Then the cursor takes in the next pair ($A$,2) and ($A$,4) till it finds an ideal pair. The subgraph matching also
starts from a node in subgraph and main graph, but for a subgraph matching we need to match all the nodes in the subgraph to the same subgraph in the data graph. After the first pair is determined the algorithm takes the next pair which includes the previous pairs plus the new pair that has been added to the list. Then the algorithm finds the matches for the group of nodes in subgraph in the datagraph. So this algorithm reduces the search space by ignoring the candidate pairs that fail to satisfy the check rules after they have been added to the partial mapping state, the whole search tree is discarded[1].

Adding C,2 and B,1 candidate pairs to the state

S1

S2

Figure 3: Adding candidate pair to state[1]

4.1.5 Complexity

The complexity is determined by the following 3 factors:
- The time taken to verify the check feasibility rules[1].
- The time taken to find the sets that relate to the new instance of state[1].
- Time taken to generate the paris which also includes the current state[1].

The number of states that are explored by algorithm while computing the candidate pairs are N. Therefore the computational complexity when selecting the candidate pairs of the state becomes O(N^2). The worst case the algorithm has to visit all the nodes and perform checks on all the candidate pairs, which results in the exhausting the whole search space excluding the candidate pairs that fail to satisfy the feasibility rules. The complexity were calculated by running the algorithm on protein datasets.

Running time (best): O(n^2) [1].
Running time (worst): O(n!n) [1]
Space: O(n^2) [1].
4.2 GraphQL

4.2.1 Overview

GraphQL algorithm is built on the concept of traditional database management systems[2]. In traditional database systems the concept of querying approach that is followed. Similarly in graphql, a subgraph that has to be matched is a query and the database is nothing but the datagraph. Find all possible mappings that are similar to the query graph in the datagraph[2]. The query graph is called the pattern and the database is called the target graph. The main goal of the graphql algorithm is to efficiently find the pattern in the large graph. On the very core level, the algorithm makes use of a dfs search. But this algorithm is rebuilt with some exciting pruning techniques, search space refinement and search order optimization[2]. These techniques have helped GraphQL to be one of the fastest algorithms.

4.2.2 Definition

Input: datagraph<\textit{G}> and pattern graph<\textit{P}>[2]
Output: one or all possible mappings between the graphs[2]
Constraints: Search order, Neighbour profile pruning, refine search space[2]
4.2.3 Design

The above figure shows the design of the algorithm, when the feasible mates are found, we can directly create a search order and then start searching. It is called normal search[2]. The neighbour profile pruning stage and Refine search space can be an addon to the algorithm, which makes the algorithm run more faster.

4.2.4 Implementation

Given a datagraph G and a pattern graph P, the main goal of the algorithm is to find one or all possible mappings of P in G. To find all the feasible mates the pattern graph is changed to predicates. A feasible mate of node $F_u$ [2] is defined as the following:

$$\phi(u) \rightarrow G[\text{verticesThatHaveLabel}][P[\text{verticesThatHaveLabel}]]$$

The GraphQL has two phases, the phase 1 finds all possible mates which creates a search space and an search order is created and the second phase consists of search algorithm that runs on the search space[2]. The search space is nothing but (feasible mates * each node in graph G)[2]. A check function ensures that a node u in P can be mapped with a node v in the G. If no then the algorithm continues to the next node, else returns true if one mapping is found.
MS Capstone Project

Input: G, P
Output: list<mappings>/ one mapping

for each node u in V(P)
    find all feasible mates.
perform Neighbour profile pruning<can be omitted>
get the search order

search(depth:0):
    for each v E phi(u),
        if check(u,v)
            phi(u) <- v
check(u,v)
    for edge belongs to E(p)
        If edge not in list of possible mates
            return false
return true

Algorithm 2: Phase 1 Graph Pattern Matching

Due to the effect on the performance of the algorithm 1 the authors define a search space refinement technique by considering 3 main things discussed below:

- How to reduce phi space and how efficiently can we retrieve from phi[2].
- Reduce overall search space[2]
- Optimizing the search order to get much faster results[2].

These three optimization techniques help the above algorithm to perform much efficiently when there are larger data graphs and different size pattern graphs[2]. For refining the search space basically we construct a bipartite graph of the pattern graph[2]. If there is a semi perfect matching then the node is unmarked else it is considered[2]. A boolean 2D matrix is used for constructing the bipartite graph. Below algorithm discusses the phase 2 of the algorithm.

Input: Pattern graph<P>, level
Output: Reduced search space

start:
    for each node in P -> marknode(u,v)
        for 1 to levelOfQueryGraph
            for all that is marked
                construct bipartite graph
                For each neighbour
                    If u has partial perfect match
                        then remove_markedNode(u,v)
                    else
                        remove node v from phi(u)
                end
        end
    If marked node u,v is not present break;

Algorithm 3: Phase 2 Refine the search space[2]
The implementation is straightforward and robust. This is one the fastest subgraph matching techniques. The algorithm can be run even without neighbour profile pruning and refining the search space which makes it work slower.

4.2.5 Example

![Example Graph G and Pattern graph G](image)

**Figure 5: Example Graph G and Pattern graph G**

**Inverse adjacency set creation:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 1: Inverse adjacency set creation*
Finding feasible mates:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2: Finding feasible mates

Neighbour profile pruning (Q difference T):

Q: (4, 0, 1)
T: (4, 0, 4, 0, 2, 0, 4, 0, 1)
Q-T = NULL
Q: (2, 1, 4)
T: (4, 0, 2, 0, 1, 4, 2, 0, 4, 4)
Q-T = NULL
Q: (2, 1, 4)
T: (1, 4, 4, 0, 2, 0, 4, 4)
Q-T = NULL
Q: (2, 1, 4)
T: (4, 0, 2, 0, 4, 0, 2, 4, 4)
Q-T = 1
Q: (2, 1, 4)
T: (1, 4, 0, 0, 4, 4, 4, 4, 0, 4)
Q-T = 2
Q: (4, 1, 0)
T: (1, 4, 1, 4, 4, 2, 0, 4, 0)
Q-T = NULL
Q: (4, 1, 0)
T: (1, 4, 4, 4, 2, 0, 4, 0)
Q-T = NULL
Q: (4, 1, 0)
T: (1, 0, 2, 0, 4, 4, 2, 4, 0)
Q-T = NULL
Q: (4, 2)
T: (1, 4, 0, 2, 0, 0, 4, 4, 0, 2, 2)
Q-T = NULL
Q: (4, 2)
T: (2, 4, 4, 2, 4, 2)
Q-T = NULL
Q: (0, 4)
T: (4, 0, 2, 0, 1, 4, 2, 0, 4, 4)
Q-T = NULL
Q: (0, 4)
T: (1, 4, 4, 4, 0, 2, 0, 4, 4)
Q-T = NULL
4.2.6 Complexity

The complexity of the algorithm can be devised by combining the complexities of the two phases which are explained above in the overview section[2]. The worst case complexity of the algorithm 1 of phase 1 which is Graph pattern matching is $O(n^k)$. But in the real world the complexity depends on the how large the search space is, as we know that the subgraph isomorphism is NP-hard[2]. Therefore in order to make the algorithm more efficient the search space refining algorithm was devised[2].
Overall: $O(\text{level} \times \text{Sum}(1..k) | \phi(u)^* (d_1, d_2 + M(d_1, d_2)))[2]$

4.3 Common Example

![Diagram](Figure 7: Common Example)

4.3.1 VF2 Working

<table>
<thead>
<tr>
<th>Candidate Pairs</th>
<th>Semantic Feasibility</th>
<th>Pred_R and Pred_S</th>
<th>RIN</th>
<th>ROUT</th>
<th>RNEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,3</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,3</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,3</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,3</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>3,2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>1,1</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,1</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>0,0</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>1,0</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Table 4: VF2 Working
4.3.2 GraphQL working

Step 1: Search Space Phi (Feasible mates)

<table>
<thead>
<tr>
<th>Query Vertices</th>
<th>Search Space PHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>1</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>3</td>
<td>[4]</td>
</tr>
</tbody>
</table>

Table 5: Search Space Phi

Step 2: Build Query map and Graph map

<table>
<thead>
<tr>
<th>Query PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DataGraph PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Table 6: Build Query map and Graph map

In the above step the highlighted row in query map and data graph map, of which the difference is taken which gives us 3. That is after pruning the following step shows the updated search space.
Step 3:

<table>
<thead>
<tr>
<th>Query Vertices</th>
<th>Search Space PHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0, 1]</td>
</tr>
<tr>
<td>1</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>2</td>
<td>[3]</td>
</tr>
<tr>
<td>3</td>
<td>[4]</td>
</tr>
</tbody>
</table>

Table 7: After NPP

Step 4: Default search order

<table>
<thead>
<tr>
<th>Search Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 0 1</td>
</tr>
</tbody>
</table>

Table 8: Default search order

Step 5: Final matched output.

<table>
<thead>
<tr>
<th>depth</th>
<th>Phi[depth]</th>
<th>Check(depth,Phi[depth])</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>2,3 Selected</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3,4 Selected</td>
</tr>
<tr>
<td>0</td>
<td>0,1</td>
<td>(0,1 )Selected</td>
</tr>
<tr>
<td>1</td>
<td>2,3</td>
<td>1,2 Selected</td>
</tr>
</tbody>
</table>

Table 9: Final matched output
5. Datasets

For testing purposes we used the biological datasets[3]. These datasets follow a simple pattern which consists of different components as shown in the below figure.

```
8,16,32,64,128,256 query sizes
T: target Node
Q: query node
N: Number of matches
S: <List> of matches
```

- backbones_1EMA.8.sub.grf
- backbones_1EE1.16.sub.grf
- backbones_1A7F7.grf
- backbones_1APM.grf

The datasets consist of ground_truth which gives all the details of target and pattern graphs. From this file the user can identify which target and pattern are to be chosen so as to run on different algorithm in the framework. This file also helps to build a generalized parser which helps to automate the matching process for a particular target and the pattern node.
6. Design

The project is divided into Algorithms, Graphs, Tests and Utility packages. Each package is used to hold different types of java files. As of now the Algorithms package consists of VF2[1] and GraphQL[2] algorithms, and the associated graph data structures are in the Graphs packages. They are different as of now, but can be built as one data structure which will make it more efficient. The main entry point of the project is the Test package where we can perform two kinds of tests, one tests all kinds of datasets that are in the datasets. The user has to enter the name of the biological dataset and the query size he wants to run (8, 16, 32...). This is automated. The results produced are matched to the ground truth file structure and the second test allows the user to run some sample examples i.e. small graphs which help to understand the algorithms. At last the Utility package consists of data parser which pre processes the data from the biological datasets and then gets it ready for the algorithms to be run on.
7. Classes and Methods

- **Algorithms**
  - BipartiteMatching()
    - Max BipartiteMatching()
  - GraphQL
    - FeasibleMates()
    - NeighbourProfilePruning()
    - RefineSearchSpace
    - SearchOrder
    - Search(depth)
  - VF2Matcher
    - match()
    - match(State,G,P)
    - Feisable()
      - Check Termin()
      - Check Terminout()
      - R_succ()
      - R_pred()
      - R_new()

- **Graph Datastructure**
  - VF2 Graph
    - Add Node()
    - Add Edge()
    - Incoming Vertices()
    - Outgoing vertices()
    - AdjMatrix()
  - GraphQL Graph
    - AddParents()
    - LabelMap()
    - LabelVertices()

- **Dataparser**
  - getDirectories()
  - bifurcateDirectories()
  - processGTFFiles()
8. Software / Hardware

Hardware:

As this project is an in memory application the hardware system specifications are as follows
- Mac OS X / Windows
- 8GB Ram
- I7 Processor

was built on this system. But it doesn't matter if the system operating system or processor is changed. The important thing is ram, this should be atleast 2GB else the performance will be slower because of recursive calls in VF2[1]. If the data graph and query size are huge and the ram is lower, it might throw out of memory exception.

Software:

- IntelliJ[6]: Platform or IDE to used to build the application.
- Java 8: one method of graphql is written in Java 8 as this version allows us to use the lambda expressions, the code looks compact.
- Maven: The project is built on maven because it helps the user to easily build the project and all the dependencies are automatically added when the project is imported into the any system, this helps the user to be carefree of building dependencies. Below pom.xml shows the dependencies that were used in the project.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.hongal.msdunston</groupId>
    <artifactId>GraphDB<br/>Query</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.graphql</groupId>
            <artifactId>graphql-java</artifactId>
            <version>18.1</version>
        </dependency>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.8</version>
        </dependency>
    </dependencies>
</project>
```

Figure 10: Maven pom.xml
• JgraphT[7]: Used as an alternative to graph creation manually
• Junit[6]: To build and run unit tests if required
• Apache Commons[8]
• Google Guava[9]: For easier building of applications. Used for file processing in this project

9. Results

9.1 Sample output

Below is the sample output of the application for a query size of 16. This output shows the time required and number of recursive calls made.

![Sample output](image.png)
9.2 Graphs

VF2[1] vs GraphQL[2] Nodes: This graph shows how much time each algorithm takes when the number of nodes increases. The graph has been plotted for 600, 1000, 3000, 5000, 7000 and 9000 nodes.

GraphQL[2] query size: This plot is taken by measuring the time when each algorithm is run against 8, 16, 32, 64, 128, 256 query size.
VF2[1] vs GraphQL[2] datasets: This graph has been plotted, by averaging 5 datasets for each query size for both.

![Graph of VF2 vs GraphQL datasets](image)

**Figure 14: VF2 vs GraphQL datasets**

10. Conclusion

- As both algorithms perform a bit differently for different kinds of query size and number of vertices in the given data graph, it is hard to identify which algorithm perform better.
- GraphQL[2] is much faster when there are smaller number of nodes and query size
- VF2[1] and GraphQL[2] are the state of art algorithms.
- VF2[1] is the only exact matching available.
- The framework built by Prof Carlos Rivero helps for making which algorithm to choose by making a fair comparison between the all the algorithms in the framework

11. Future Work

- Focus on join order and pruning techniques
- As subgraph matching is NP-hard problem, there is still lot of scope for research in this field.
- VF2[1] plus can be integrated into framework, this helps to better compare between all three versions of VF[4] which are VF[4], VF2[1], VF2 plus[5].
12. References


3. https://thebiogrid.org/


8. https://commons.apache.org/