Marginal improvements to VF3 in its implementation to determine subgraph isomorphism between two graphs

Grishma Saparia

Abstract

The ability to determine if a smaller graph is contained within a greater graph has become increasingly important. From comparing compositions of different molecular structures to the relationships between millions of users in social media, being able to solve this time-consuming problem in as little time as possible is challenging. Among the various implementations to solve this problem is VF3, a state-of-the-art graph-matching algorithm designed by Carletti et al. We propose some improvements to its runtime by making modifications to how nodes labels are searched and stored during the initial stages of the algorithm and compare our results to Carletti's. While our improvements did not show a significant gain in time, the presence of some gain in time might be indicative of a greater improvement.

1. INTRODUCTION

1.1 Background

One of the utility features of graphs that have made them so ubiquitous in data representation is that graphs can be used to map the relationships of real-world objects or networks. These graph-mapped relationships can be examined further for the purposes of pattern recognition between the nodes of a single graph, or between the nodes of several graphs. Pattern recognition is an important concept that drives many real-world technologies such as GPS, image processing and even social media networks. The backing algorithms that allow for many pattern-recognition and pattern-matching technologies to function, often solve the problem of subgraph isomorphism.

However, solving this problem is very time-consuming. As the number of nodes and their connections increase, the time to verify if one graph is contained within another increases exponentially. Several approaches to this problem have been proposed over the years, each with their own strengths and drawbacks depending on the properties of the graphs being analyzed. Even though computing power in increasing on a yearly basis, efficient ways to solve this problem are still desirable.

2. Problem Definition

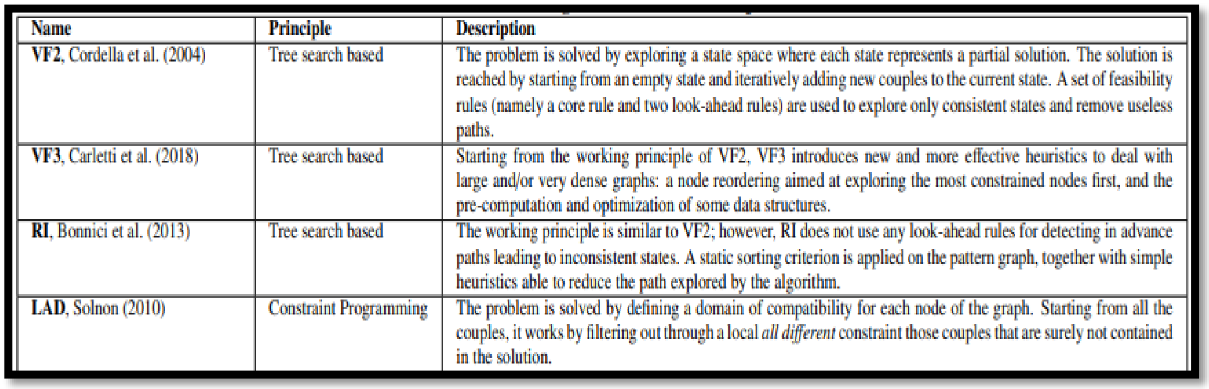
2.1 Definitions

Graph isomorphism is a check of similarity between two different graphs, whereas subgraph isomorphism identifies whether an input graph is a part of the larger graph. In solving the subgraph isomorphism problem, researchers have extended the solutions of the problem to applications in various fields such computer design systems, social networks and etc. For example, the Vertex and Edge Approximate graph Miner (VEAM) algorithm solves the problem of subgraph isomorphism in order to aid in tasks that require image processing through matching various image segments [3]. Such algorithms are highly applicable to forming analysis on objects and networks represented as nodes and edges.

The subgraph isomorphism computational problem proposes, given two input graphs G and H, determine whether or not the target graph, G contains a subgraph that is isomorphic to the pattern graph, H. A formal mathematical definition of subgraph isomorphism is as follows; A Graph G = (N,E) consists of n node and edge set $E \subseteq N \times N$, where edges (u, u’). A Subgraph isomorphism problem between a pattern graph Gp = (Np, Ep) and target graph Gt = (Nt, Et) is whether Gp is isomorphic to some subgraph of Gt [2]. One should find an injection matching $f : Np \rightarrow Nt$, such that the association between a different target node to each pattern node preserves the pattern edges, $\forall (u, u') \in Ep \; (f(u), f(u')) \in Et$. The function f is called a sub isomorphism function [2].

2.2 Present Solutions

There are various algorithmic approaches to solving the subgraph isomorphism problem, the more prominent algorithmic approaches involve either constraint programming, tree-based searching, or graph indexing. The tree search approach has been shown to be the fasted approach to solving the subgraph isomorphism problem [1]. Tree searching obtains solutions by searching inside a state space through depth-first search with backtracking. The solutions are determined incrementally at each new state [1]. Depth-first search does not require all the states to be stored in memory, the maximum number of allocated states is proportional to the number of nodes, therefore the space for each state is constant [1]. The more prominent tree search-based algorithms for solving subgraph isomorphism problems include Ullmann, VF2, RI, and VF3 [1]. These algorithms differ in the heuristic rules used for pruning states not leading to solutions [1]. The tradeoff for these tree-based algorithms is between the effectiveness of the pruning produced by the rules and the computational cost required for their implementation [1].



Name	Principle	Description
VF2, Cordella et al. (2004)	Tree search based	The problem is solved by exploring a state space where each state represents a partial solution. The solution is reached by starting from an empty state and iteratively adding new couples to the current state. A set of feasibility rules (namely a core rule and two look-ahead rules) are used to explore only consistent states and remove useless paths.
VF3, Carletti et al. (2018)	Tree search based	Starting from the working principle of VF2, VF3 introduces new and more effective heuristics to deal with large and/or very dense graphs: a node reordering aimed at exploring the most constrained nodes first, and the pre-computation and optimization of some data structures.
RI, Bonnici et al. (2013)	Tree search based	The working principle is similar to VF2; however, RI does not use any look-ahead rules for detecting in advance paths leading to inconsistent states. A static sorting criterion is applied on the pattern graph, together with simple heuristics able to reduce the path explored by the algorithm.
LAD, Solnon (2010)	Constraint Programming	The problem is solved by defining a domain of compatibility for each node of the graph. Starting from all the couples, it works by filtering out through a local *all different* constraint those couples that are surely not contained in the solution.

Pictured: State of the art algorithms used to solve subgraph isomorphism

The algorithm that we focused on researching, implementing, and improving upon, is a tree-based searching algorithm developed by Italian researchers in 2018, called VF3. VF3 is an algorithm for subgraph isomorphism formulated to deal with huge graphs, still problematic for most of the state-of-the-art algorithms [1]. Even the well-run algorithms have a practical limit of applicability for the graphs with few hundreds of nodes. Graph density, i.e., the ratio between the number of edges of the graph and the maximum number of edges in a graph having the same number of nodes, is also one of the factors influencing the complexity of the matching [1]. In fact, sparse graphs i.e., graphs with low density and if the degree is bounded by a constant, there are algorithms that solve in polynomial time. On the contrary, graphs which are both dense and large still poses a significant difficulty to matching algorithms [1]. Research on VF3 proves that it is the fastest algorithm for large and dense graphs, and that its matching time is close to the best [1]. Furthermore, the VF3 algorithm shows a reduced variance in matching time with respect to other algorithms, in particular RI [1]. As a result, researchers conclude that, "VF3 is a safe choice for applications where the characteristics of the graphs are not completely known in advance. [1]" Hence, in our implementation we proposed an improvement to the time efficiency of the VF3 algorithm.

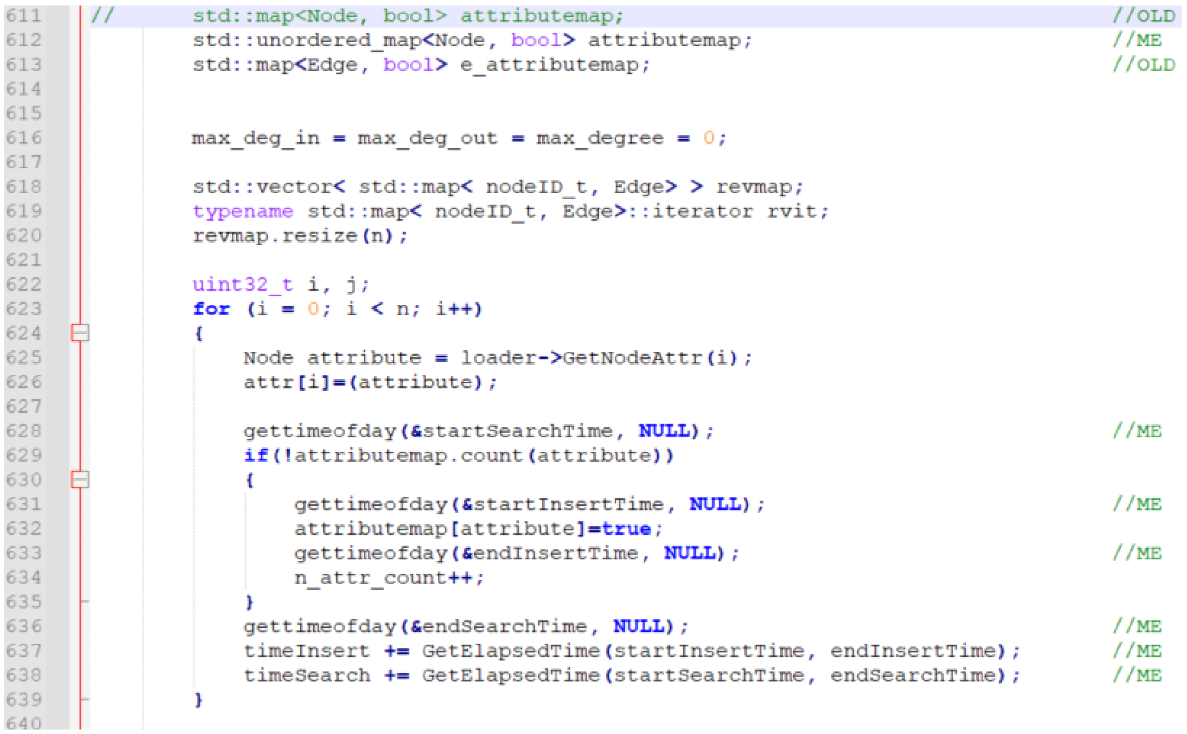
While VF3 is equipped to deal with both labeled and unlabeled graphs, in our specific tests we looked strictly at labeled graphs. This is due to the fact that, since we are measuring the times for lookup and storage of labels, there would have been nothing to measure on unlabeled graphs.

3. Proposed Solution

3.1 Implementation Details and its Time Complexity

After analyzing the code used by Carletti et al [4], we believe we located a source of improvement. As the input file is being processed, it stores all the different labels it finds associated with the nodes in a map. Despite its name, the implementation of a map in the C++ standard library is done using a red-black tree [5], which has logarithmic time to all of its dictionary operations. We believed we could improve on it by instead using an unordered map, which is implemented in the C++ standard library as an open-hash array [6]. Such an array has a constant time-complexity for all dictionary operations on average, and linear in its worst case.

The specifics can be seen in figure below. The unordered_map library is imported inside the ARGraph library, and the data structure for storing the node attributes is changed from map to unordered_map. We then added code which would allow us to measure how long it took for both searching and inserting attributes found in the input file. While we do not expect to see a big reduction in the overall runtime of VF3 (the pattern-matching step is the algorithm's biggest source of time complexity), we expected to see some improvement in time-complexity as the nodes are being initialized.



```
//      std::map<Node, bool> attributemap;                                        //OLD
        std::unordered_map<Node, bool> attributemap;                              //ME
        std::map<Edge, bool> e_attributemap;                                      //OLD


        max_deg_in = max_deg_out = max_degree = 0;

        std::vector< std::map< nodeID_t, Edge> > revmap;
        typename std::map< nodeID_t, Edge>::iterator rvit;
        revmap.resize(n);

        uint32_t i, j;
        for (i = 0; i < n; i++)
        {
            Node attribute = loader->GetNodeAttr(i);
            attr[i]=(attribute);

            gettimeofday(&startSearchTime, NULL);                                 //ME
            if(!attributemap.count(attribute))
            {
                gettimeofday(&startInsertTime, NULL);                             //ME
                attributemap[attribute]=true;
                gettimeofday(&endInsertTime, NULL);                               //ME
                n_attr_count++;
            }
            gettimeofday(&endSearchTime, NULL);                                   //ME
            timeInsert += GetElapsedTime(startInsertTime, endInsertTime);         //ME
            timeSearch += GetElapsedTime(startSearchTime, endSearchTime);         //ME
        }

```

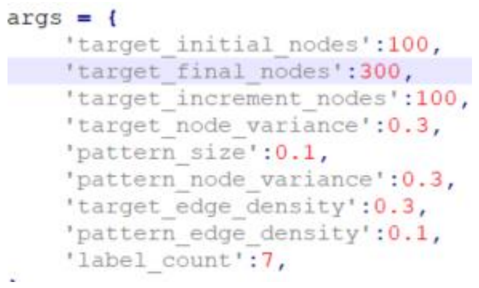
Pictured: Proposed improvements made to the code. Lines with the comment “//ME” shows code which was added by us.

3.2 Generating the Graphs

Since we were unable to acquire the labeled graphs used by Carletti et al (only the unlabeled graphs were available), and we were having trouble using the generators provided (we could not locate a few of the libraries necessary to compile them), we devised our own generator. Our generator creates random directed connected graph pairs: One smaller graph, the pattern, and one bigger graph, the target.

As a relatively simple Python script, our generator takes the following as input: number of nodes in the target graph, the percentage of variance in the number of nodes in the target graph, the number of nodes in the pattern graph relative to the target graph, the variance in the number of nodes in the pattern graph, the edge density in both graphs, and the number of labels present in both graphs. To help automate the generation graphs of different sizes, an initial, final, and step inputs are also accepted, which dictate how big the smallest target graph should be, how large the largest one should be, and the difference in nodes between the current graph and the next one to be generated.

As an example, let’s assume the generator is passed the input in figure to the right. Since our initial number of nodes in the target graph is 100 and the final is 300, with a step increment of 100, 3 graphs will be generated: 100, 200, and 300. Let’s analyze the one with 100 nodes.



```
args = {
    'target_initial_nodes':100,
    'target_final_nodes':300,
    'target_increment_nodes':100,
    'target_node_variance':0.3,
    'pattern_size':0.1,
    'pattern_node_variance':0.3,
    'target_edge_density':0.3,
    'pattern_edge_density':0.1,
    'label_count':7,
```

Pictured: A simple example of the types of inputs the generator takes

Having the number of nodes set at 100, the generator then calculates the variance of nodes in this graph. In this example, it’s 30% (0.3), meaning this graph could have any number of nodes between 70 or 130. A random number is then picked from this interval, which will be this graph’s final node count.

Next, the same is done for the pattern graph. The size of the pattern graph isn’t an integer, but a percentage of the target graph. In our example, it is 10% (0.1), meaning our pattern graph will have 10 nodes. The variance of the pattern graph is also calculated, which would yield a range between 7 and 13. A random number is picked from this interval.

Next, the density of the edges must be calculated, which is based on a percentage of the number of edges an equivalent complete graph would have. A complete graph is simply a graph where every node is connected all other nodes, meaning this graph has the maximum number of edges possible. This can be calculated through the formula **(nodes*(node-1))/2**. The density of the edges in our graphs is calculated by multiplying the percent density by the maximum number of possible edges. In our example, our target graph of 100 nodes can have a maximum of 4950 edges; since our density multiplier is 30% (0.3), the final number of edges in our target graph is 1485. Doing the same for our pattern graph yields a total of 4.5 edges. These numbers are always rounded down and divided by the number of nodes so that each node has an equal number of edges. If such a division means that nodes would have less than 1 edge, then each node will have 1 edge.

Now that the exact number of nodes and edges in both graphs is known, we can start assembling the graph: For each node in both graphs, we assign the number of edges that node could have. Since our graphs are directed, we only need to decide what the target of the edge will be, since all nodes will have at least one edge, and their source is themselves. The target of the edge is picked randomly. Note that a node cannot have an edge pointing to itself, nor it can have two edges pointing to the same other node. We use a simple list to keep track of illegal destinations. All nodes are also assigned a label, an integer from 0 to the maximum defined in the input (in our example, from 0 to 6)

Finally, the pair of graphs generated needs to be validated for isomorphism, otherwise they cannot be used. This is done by simply feeding both graphs to our benchmark and verifying if a solution exists. If it does, the graphs are saved, otherwise, a new pair is generated.

It’s worth pointing out that no guarantee can be made that the graphs are isomorphic, since their number of nodes, edges, the destination of each edge, and the label in each node is randomly selected, hence why the pair must be validated before being saved. The higher the relative size of the pattern graph, its edge density, and its number of labels compared to the target graph, the harder it will be to generate a match.

4. Performance Evaluation

4.1 Environment and Preparation

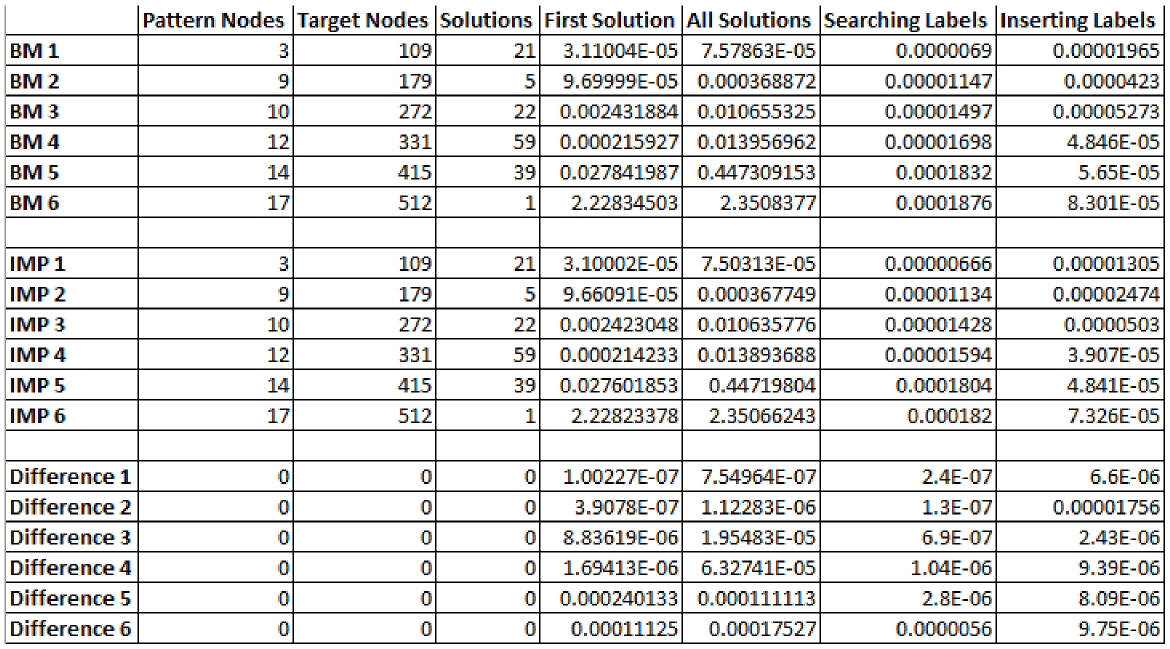
Both our graph generation algorithm, the benchmark, and our improved version of the benchmark were running on an AMD Ryzen 5 2600x six-core processor with 16 Gb of RAM and 100 GB of SSD disk space. The operating system is Ubuntu 20.04.4 LTS (x64), while our software suite includes python 3.8.10, and G++ 9.4.0. As VF3, we are running the master branch which can be found at Carletti's GitHub repo [4]. Specifically, we are running the parallel version of VF3, as it allows us to make full use of our multicore processor. The graph generation itself took 14 hours, while the matching for both the benchmark and the improvement took 30 minutes in total.

We generated 12 graphs, which range from 100 to 600 nodes for target graphs (in steps of 100) and a size of 5% for pattern graphs; a node variance of 30% for both graphs; edge density of 30% and 10% for target and pattern, respectively; finally, our number of labels was 7 for both graphs.

We selected 7 on purpose, a red-black tree with 7 different entries would have a depth of 3. Indeed, the higher the depth of said tree, the longer it will take to find an entry in it, on average. However, while we expected to see a greater difference in dictionary operations the more labels we used between the open-hash and the red-black tree, a higher number of labels would increase the chances of generated graphs not being isomorphic, as there would be a greater mismatch between the labels assigned to both graphs.

Finally, we ran both the benchmark algorithm and our improvement on each pair of graphs 100 times and took the mean average of all the results. Since our version of Linux is a common time-sharing operating system, our algorithms needed to share their use of the computer's processor with other running processes. Despite running everything on a fresh installation, no guarantees can be made that no other processes would preempt our code, hence the large number of trials per graph-pair. Running these algorithms on a real-time environment should more accurately show their actual runtime.

4.2 Results

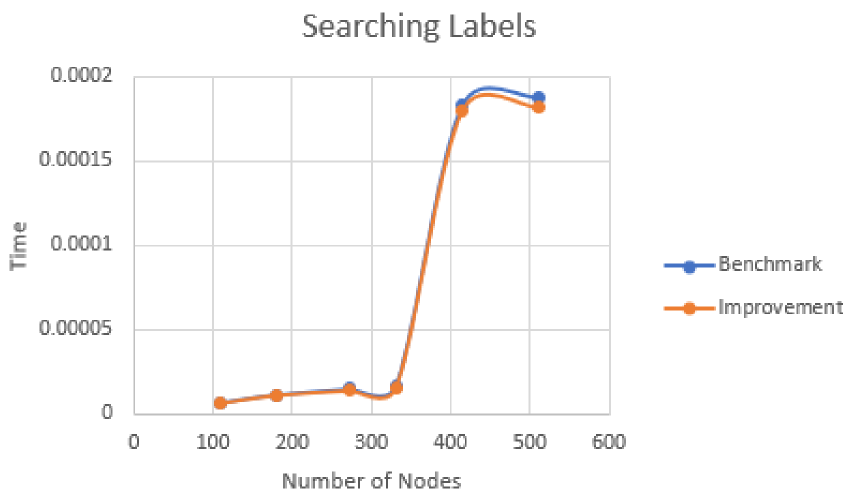


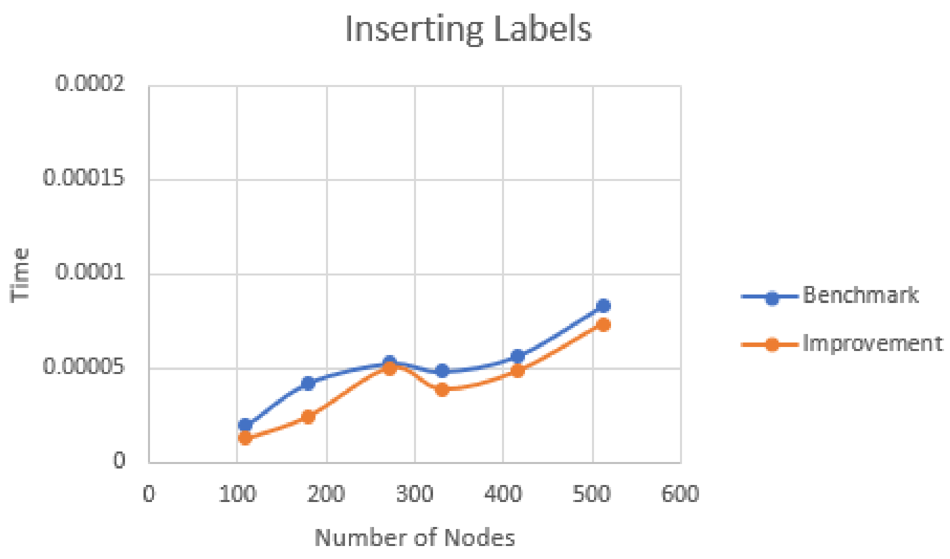
	Pattern Nodes	Target Nodes	Solutions	First Solution	All Solutions	Searching Labels	Inserting Labels
BM 1	3	109	21	3.11004E-05	7.57863E-05	0.0000069	0.00001965
BM 2	9	179	5	9.69999E-05	0.000368872	0.00001147	0.0000423
BM 3	10	272	22	0.002431884	0.010655325	0.00001497	0.00005273
BM 4	12	331	59	0.000215927	0.013956962	0.00001698	4.846E-05
BM 5	14	415	39	0.027841987	0.447309153	0.0001832	5.65E-05
BM 6	17	512	1	2.22834503	2.3508377	0.0001876	8.301E-05
IMP 1	3	109	21	3.10002E-05	7.50313E-05	0.00000666	0.00001305
IMP 2	9	179	5	9.66091E-05	0.000367749	0.00001134	0.00002474
IMP 3	10	272	22	0.002423048	0.010635776	0.00001428	0.0000503
IMP 4	12	331	59	0.000214233	0.013893688	0.00001594	3.907E-05
IMP 5	14	415	39	0.027601853	0.44719804	0.0001804	4.841E-05
IMP 6	17	512	1	2.22823378	2.35066243	0.000182	7.326E-05
Difference 1	0	0	0	1.00227E-07	7.54964E-07	2.4E-07	6.6E-06
Difference 2	0	0	0	3.9078E-07	1.12283E-06	1.3E-07	0.00001756
Difference 3	0	0	0	8.83619E-06	1.95483E-05	6.9E-07	2.43E-06
Difference 4	0	0	0	1.69413E-06	6.32741E-05	1.04E-06	9.39E-06
Difference 5	0	0	0	0.000240133	0.000111113	2.8E-06	8.09E-06
Difference 6	0	0	0	0.00011125	0.00017527	0.0000056	9.75E-06

The table above shows the raw results of our trials, while the graphs in the next page show their plotted values.

For the table, each column represents a different metric which was tracked, while the rows represent the different trials. BM stands for benchmark, the original implementation of VF3 by Carletti. IMP stands for improvement, which is our implementation. Difference shows the difference in output between Carletti's code and ours. A positive number means our improvement beat Carletti's by the indicated amount, while a negative number means the opposite.

For the graphs, the X-axis shows the number of nodes in the target graph, while the Y-axis shows how much time, in seconds, that operation took to run. Thus, "Searching Labels" shows how long the computer spent verifying if the label of the node it was currently evaluating had already been seen, while "Inserting Labels" shows how long the computer spent adding new labels to its data structure (map and unordered_map).





4.3 Discussion

Regrettably, while a small improvement could was observed when retrieving existing labels which had already been processed, the difference is small enough to not be of any notice. More interestingly, however, is the different in insertion times between the benchmark and the improvement; it is still small, specially considering the time scale (the differences come mostly down to microseconds), but noticeable. It is also worth noting that the overall time for finding the first solution and finding all solutions improved, even if only on a scale of microseconds. We have a few hypotheses on the reason for such a small improvement.

The open-hash itself could be at fault during runtime. While it has a constant time-complexity for all dictionary operations on average, it has a linear time-complexity on worse cases. This means that if many of the labels assigned to it fell into the same bucket, the computer would still need to find the label it is looking for in the list inside that bucket, which would be done in linear time. Also, it is not clear to us if the size of the open-hash's array is initialized with a size big enough to accommodate all labels individually. One strategy to improving an open-hash's access time is to keep track of the current depth of the deepest bucket; if it deviates too much from the other buckets, one should double its array's size and re-hash all of its contents. This computation would still save time as the array gets larger and more elements are added, but for trials with such small timeframes as ours, it could have been a detriment.

Another reason for the small difference might have been in our input sizes. While some of our graphs would be considered of medium size according to Carletti, maybe greater gains would only start being observed as input sizes reach the 1000s. Our pattern-graph sizes and number of labels used are also comparatively very small, which could have impacted the results.

Yet another reason might lie in our graph generator. Indeed, our method of generating graphs is very random, but uniform. All nodes have the same number of outgoing edges, but a random number of incoming edges. This does not allow VF3 to fully utilize its heuristics: if there is little to no variance in the cardinality of all nodes (i.e. the number of edges incoming and outgoing), then classification must be done primarily through the labels; and that failing, then very few state-spaces will be pruned prior to the search.

5. Prognosis

We were fortunate to be able to conclude all of our milestones within our deadline. We identified a problem we wanted to tackle, analyzed its literature for implementations, analyzed a candidate implementation for improvement, implemented said improvement, and compared it to the original. Planning our timeline with more time than we believe we needed gave us the flexibility to take our time on each step, while also giving us concrete deliverables for which to strive. The whole process gave us a better understanding of how to conduct this kind of research in the future, as well as what difficulties to expect.

Among some of the difficulties encountered by our team was in analyzing the code itself. Our team being mostly familiar with Java and Python, we struggled as we scrambled to learn C++, so we could understand how things were being implemented. Despite C++ being object-oriented, its use of pointers complicated our understanding. At one point, we considered both rewriting everything from scratch in Java and using some product or service to convert the code from C++ to Java; the first plan was scrapped due to our time constraints, as was the second since we did not find a solution with which we were happy.

A few other noteworthy difficulties included the loss of two of our team members (they didn't die, they simply left the project) and our initial candidate improvement turning out to be meaningless, as the portion of the code being considered for improvement wasn't being used at all during compilation. We discuss this further down in the conclusion.

It was particularly interesting relating a program's theoretical time-complexity to its actual run-time. It is one thing to calculate that a program's runtime grows exponentially in regards to its input size, it is another to watch a computer take twice as many hours to calculate something when you double the input. The waiting makes it painfully clear why analysis of a program's time-complexity is so important. And while we may be spoiled as computers increase in their processing and storage capacity over the years, it is important to remember that raw computational power might justify sloppy solutions for small inputs, but it will not save you as the problem grows in size.

6. Conclusion

While it is regrettable that we did not observe as much of a time save when comparing our proposed implementation to VF3's original implementation as we hoped, we believe our improvements show the possibility of greater gains as both input sizes and edge density increase. Our method of generating new graphs was also not ideal, taking much of the time we could have spent running trials.

For future work, we consider two things: tidying up the codebase and looking for other sources of improvements.

In our analysis of the codebase, we encountered several files which did not seem to impact the runtime at all, or indeed were not even being compiled. We made several changes to these files, yet none were being reflected during execution. Our conclusion was that these files were artifacts of previous versions, left behind for historical purposes. These made analyzing the code difficult, as our assumptions of what improvements we could do kept changing as we realized the section we were analyzing wouldn't actually run.

We believe it is also worthwhile to further explore the codebase for other small sources of improvements. Given that subgraph isomorphism is such a computationally challenging problem as the input grows (which can take from hours to days for a solution to be found), careful analysis of any third-party libraries used might reveal further sources of improvement in the runtime.

References

[1] V. Carletti, P. Foggia, A. Saggese and M. Vento, "Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 40, no. 4, pp. 804-818, 1 April 2018, doi: 10.1109/TPAMI.2017.2696940.

[2] Christine Solnon. AllDifferent-based Filtering for Subgraph Isomorphism. Artificilligence, El-sevier, 2010, 12-13, 174, pp.850-864. 10.1016/j.artint.2010.05.002 . hal-01381476

[3] Somkunwar, R., & Moreshwar, V. (2017). A comparative study of graph isomorphism applications. International Journal of Computer Applications, 162(7), 34–37. https://doi.org/10.5120/ijca2017913414

[4] vf3lib: https://github.com/MiviaLab/vf3lib, retrieved on May 2022

[5] std::map https://en.cppreference.com/w/cpp/container/map, retrieved May 2022

[6] std::unordered_map https://en.cppreference.com/w/cpp/container/unordered_map, retrieved May 2022

[7] We would also like to thank Ryan Deem and Arshdeep Sandhu, for their contributions during the initial stages of the project