A Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs

by

Matthew Wyatt Saltz

(Under the direction of John A. Miller)

Abstract

In recent years, the robustness of graphs for representing complex data has led to a proliferation of research on graph databases and analytics. One important topic in the field is graph pattern matching, which can be used, for example, in the processing of queries in graph databases. Though fast querying is highly desirable, pattern matching algorithms are hindered by the NP-completeness of the subgraph isomorphism problem. This paper presents a conceptually simple, memory-efficient, pruning-based algorithm for the subgraph isomorphism problem that outperforms commonly used algorithms by orders of magnitude on large labeled graphs. This speedup is due in large part to the effectiveness of the pruning algorithm, known as dual simulation, which in many cases removes a large percentage of the vertices not found in isomorphic matches. In this paper, the runtime of the algorithm is tested on synthetic graphs of up to 10 million vertices and 250 million edges and on two real life datasets, comparing when possible to an adjacency list version of Ullmann's algorithm and to the VF2 algorithm. To the best of our knowledge, this is the first paper to test a centralized subgraph isomorphism algorithm on graphs of this magnitude. The algorithm is tested extensively to determine the effects of label density, edge density, data

graph size, degree distribution, and query graph size and type on runtime. The effectiveness of the algorithm is then demonstrated on two large real life graphs. The algorithm is easily extendable to graphs with multiple attributes on vertices and edges, making it an ideal candidate to serve as the backbone of a query processing engine for a graph database.

Index words: pattern matching, graph, subgraph isomorphism, query processing

A Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs

by

Matthew Wyatt Saltz

B.S., University of Georgia, 2013

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2013

©2013

Matthew Wyatt Saltz

All Rights Reserved

A Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs

by

Matthew Wyatt Saltz

Approved:

Major Professor: John A. Miller

Committee: Walter D. Potter
Lakshmish Ramaswamy

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2013

A Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs

Matthew Saltz

Acknowledgments

I'd like to thank Dr. Miller, Dr. Potter, and Dr. Ramaswamy for all of their help and support throughout the research process. I'd especially like to thank Dr. Miller for all of the time he put in to help me think through my algorithms and their correctness, guide me through the research process, and edit the paper. I've really learned a lot. I'd also like to thank all of my friends and family for their support through many late nights of research. I'd like to give a special thanks to Arash Fard and Usman Nisar for their guidance throughout the process. I wouldn't be where I am without all of you.

Contents

Acknowledgments **v**

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

2 Background & Terminology **3**

2.1 What Is a Graph? 3

2.2 Graph Pattern Matching 5

3 Techniques for Graph Pattern Matching **9**

3.1 Ullmann's Algorithm 10

3.2 VF2 Algorithm 11

3.3 Related Work 11

4 Presentation of Algorithm **14**

4.1 Retrieval of Feasible Matches 14

4.2 Pruning Algorithms 15

4.3 Dual-based Isomorphism 20

5 Experimentation **25**

5.1 Synthesized Data 27

5.2 Real Data 31

5.3 Effectiveness of Dual Simulation 33

6 Conclusion & Future Work **35**

Bibliography **41**

Appendix A Instructions for Download and Execution of Source Code **42**

List of Figures

2.1 A possible social graph. 4

2.2 A possible Facebook graph query. 5

2.3 An example of a subgraph isomorphism query (left) and its matches in a data graph (right). 7

4.1 An example of a query graph (left) and its matches via simple, dual, and isomorphism in a data graph (right). Simple matches are denoted by an ‘s’, dual matches by a ‘d’, and isomorphic matches by an ‘i’. Note that any vertex that matches via isomorphism also matches via dual, which in turn matches simple. 16

4.2 Pseudocode for the graph simulation algorithm 16

4.3 Pseudocode for the dual simulation algorithm 19

4.4 Pseudocode for the dual-based isomorphism algorithm 24

5.1 The average runtimes of 10 queries for each graph size indicated on 5 of each kind of graph. 28

5.2 The average runtimes of 10 queries for each query size indicated on each of 5 different randomly generated graphs. Ullmann’s algorithm is not displayed in the large BFS queries because it took a prohibitively long time in many cases. 29

5.3 The average runtimes of 10 queries for each number of labels indicated for each of 5 different randomly generated graphs, $|V| = 10^4$. 30
5.4 The effect of graph density on runtime. 31
5.5 The average runtimes of 100 queries of each size on the Youtube dataset. . . 32
5.6 The average runtimes of 100 queries of each size on the WordNet dataset. The times are shown only for the first 1024 results, if necessary. 33

List of Tables

2.1 Matches 6

Chapter 1

Introduction

Due to their ability to represent a wide variety of problems, graphs have been widely studied in both theoretical and practical contexts for decades [9, 2, 16, 25]. The bonds between atoms in molecules [13, 14, 22], the links between web pages on the Internet [21], and the relationships between members of a social network [34, 7] are all classic examples of real world graphs. Because graphs provide such an intuitive way to model so many kinds of data, many researchers have focused on developing techniques for their storage and analysis, and with the increasing importance of graph databases, efficient query processing on graphs has become an important topic of research with many applications [19, 25, 30, 36, 31, 35, 11, 32]. Query processing differs between two principle kinds of graph databases [16, 25]. In the first kind, which consists of a collection of small to medium size graphs, query processing involves finding all graphs in the collection that are similar to or contain similar subgraphs to a query graph. In the second case, which we focus on in this paper, the database consists of a single large graph, and the goal of query processing is to find all of its subgraphs that are similar to the given query graph. Both cases, however, reduce to the subgraph isomorphism problem, which is NP-complete [9], meaning that exact query matching is intractable for large graphs in the worst-case. For this reason, fast algorithms for subgraph isomorphism on large graphs

are highly desirable.

Though some heuristic algorithms for matching have been developed to avoid the combinatorial worst-case time complexity of subgraph isomorphism [8, 33, 26, 27], in practice, exact subgraph matching can often be very feasible when dealing with labeled graphs and limited size queries, even if the graphs are large. Ullmann's algorithm [32] and the VF2 algorithm [11] are two examples of widely-used subgraph isomorphism algorithms. Though in certain cases these algorithms take prohibitively long amounts of time for query processing purposes, they can be effectively used for labeled graphs with thousands of vertices.

In this paper we discuss the landscape of graph pattern matching, and we present our own memory-based algorithm for subgraph isomorphism that outperforms commonly used algorithms in many cases on large labeled graphs. The algorithm is similar to other tree search based techniques, but we use a simple pruning technique that is shown to drastically reduce the search space, often eliminating most vertices not contained in some subgraph isomorphic match. We quantify this with extensive experimentation on both synthetic and real graphs of up to 10 million vertices and 250 million edges, comparing runtimes, when possible, with an adjacency list version of Ullmann's algorithm and with the VF2 algorithm. To the best of our knowledge, this is the first paper to present results for any centralized subgraph isomorphism algorithm on graphs of this size. We discuss the major factors that influence the runtimes of the algorithms, and we explain why, and in what cases, exact matching can be practical for query processing on large graphs. Chapter 2 presents background information on the subgraph isomorphism problem and explains the terminology used in the rest of the paper. Chapter 3 discusses the basics of graph pattern matching and discusses two principle subgraph isomorphism algorithms. In Chapter 4, we present our algorithm for subgraph isomorphism, along with the pruning algorithm used to reduce the search space, and we describe its time complexity. We present experimental results in Chapter 5, and we conclude with a summary of the results presented and a discussion of future research opportunities.

Chapter 2

Background & Terminology

This chapter begins by covering the basics of graphs and the related terminology used in the paper, before describing graph pattern matching and the subgraph isomorphism problem.

2.1 What Is a Graph?

Conceptually speaking, graphs are data structures that provide an intuitive way to model various entities and the relationships between them. For example, a social network like Facebook can easily be modeled as a graph, where each member or page is an entity and the relationships between them are "friends", "likes", and so on. An example of this can be seen in Figure 2.1. In graph terminology, each of these entities is known as a *vertex*, and each of these relationships is known as an *edge*. In this example, the edges are *directed*, which is indicated by the arrows in the figure, but there are also *undirected* graphs, with directionless edges. The names of the vertices and the edges are called *attributes*, which we refer to as *labels*. This paper discusses only directed graphs with natural number vertex labels, no edge labels, and no multiple edges. Thus, within the context of this paper, a graph is a triple $G(V, E, l)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges (where the first vertex

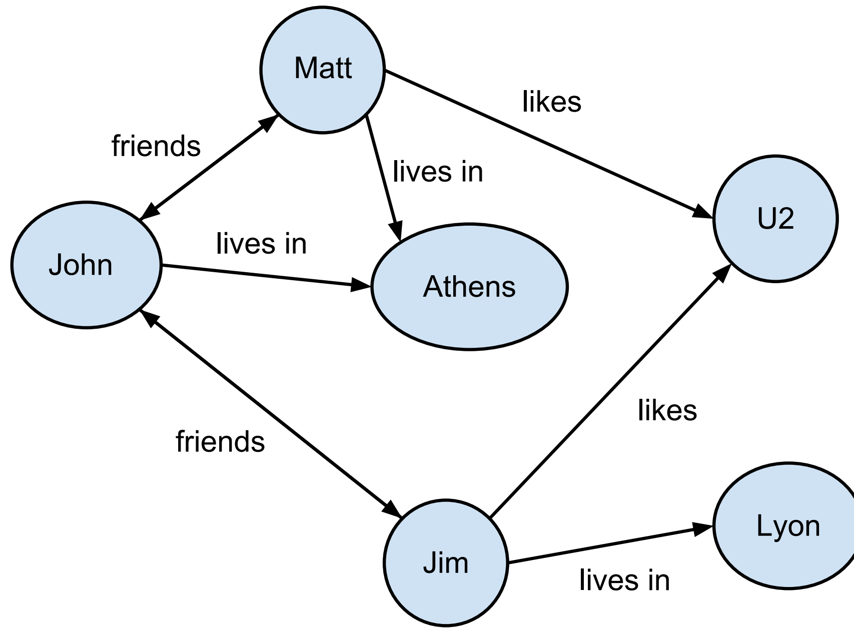


Figure 2.1: A possible social graph.

indicates the "from" vertex, and the second vertex indicates the "to" vertex), and $l : V \to \mathbb{N}$ assigns a label to each vertex. For a given vertex $v \in V$, we use $adj(v)$ (short for adjacency set) to denote the set of vertices to which v has an outgoing edge. In other words, for some $v \in V$, $adj(v) = \{v' : (v, v') \in E\}$. We sometimes refer to the vertices in $adj(v)$ as the *children* of vertex v, and conversely, we refer to v as the *parent* of all vertices in $adj(v)$. We use the term *degree* to refer to the outdegree of a vertex, which is the size of its adjacency set.

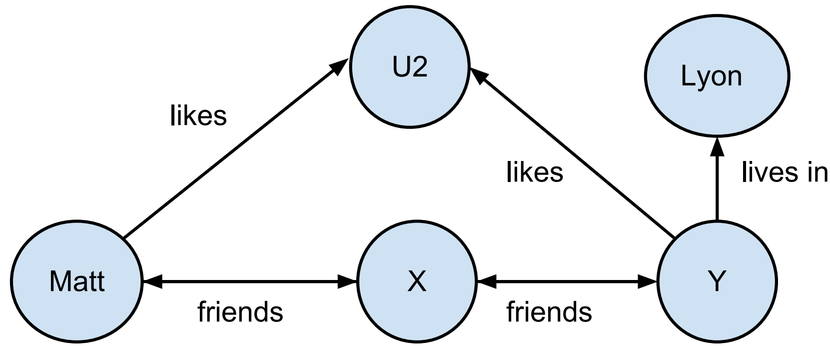


Figure 2.2: A possible Facebook graph query.

2.2 Graph Pattern Matching

The goal of graph pattern matching is to find all subgraphs of a large graph, called the data graph, that are similar to a query graph structurally and semantically.[1] To consider why this might be useful, look again at the graph displayed in Figure 2.1. Suppose a user of Facebook has just moved to Lyon, France from the United States and is looking for someone to accompany him to the U2 concert there. Thus, he wants to find out if any of his friends are friends with a person who lives in Lyon and who likes U2. Given that Facebook can be modeled as a graph like Figure 2.1, this question can be modeled as a graph query, as shown in Figure 2.2. In this query, the capital letters represent variables that the user would like filled in. In this case, using the graph in Figure 2.1, the query would return $X = John$ and $Y = Jim$. In fact, Facebook Graph Search[2] attempts to allow users to do just that, though at the moment it only has limited querying capabilities. Although this paper does not treat query processing systems with wildcards and edge attributes, the algorithms presented serve as the backbone for more general systems such as these.

[1] For our purposes, semantic matching just refers to the consideration of vertex and edge attributes when determining the similarity of two graphs.

[2] http://www.facebook.com/about/graphsearch

Table 2.1: Matches

Query Vertices	Match 1	Match 2
0	4	4
1	1	1
2	5	5
3	2	6

2.2.1 Exact vs. Inexact Matching

There are two general categories of matching algorithms: exact and inexact [2]. The similarity of two graphs in exact matching is measured on a binary scale; either the two graphs match, or they do not. Exact matching seeks to find all embeddings of a query graph in a data graph with the same structure and labelings. This similarity metric is based on subgraph isomorphism, which is defined in a moment. In inexact matching, similarity measures besides isomorphism are used that allow more leniency when matching. Some use a cost function like edit distance to obtain a real value indicating how different two graphs are [2, 6, 10, 29]. These algorithms find the matching subgraphs with the lowest cost (or highest similarity values). Some of these algorithms are still guaranteed to find all exact matches if they exist, but retain an exponential worst-case time complexity [9, 16]. Others reduce the worst-case time complexity to polynomial time at the expense of accuracy [9, 16, 25, 8, 33]. There are also algorithms which find reasonable, but not exact, matches by using heuristics without necessarily using a similarity function [26, 27]. This paper focuses on exact matching, and specifically, algorithms for subgraph isomorphism, and thus, will not discuss inexact matching further.

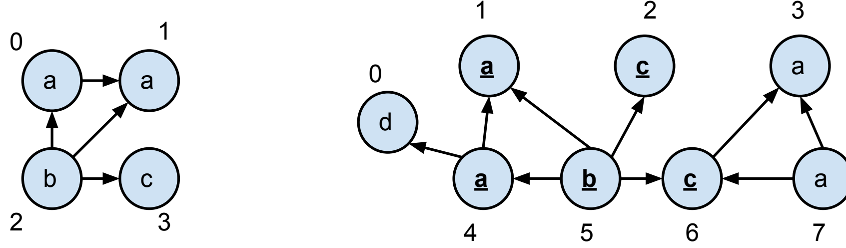


Figure 2.3: An example of a subgraph isomorphism query (left) and its matches in a data graph (right).

2.2.2 Subgraph Isomorphism

A common way to define the criteria for exact matching is to use the definition of subgraph isomorphism. Conceptually, a graph is subgraph isomorphic to another if the first graph matches a subgraph of the second structurally and semantically. A query graph and its subgraph isomorphic matches from a data graph are shown in Figure 2.3 and Table 2.1. Technically, we define *subgraph* and *subgraph isomorphism* as follows:

Definition 2.1. *(Subgraph) Given a graph $G(V, E, l)$, a graph $G'(V', E', l')$ is said to be a* subgraph *of G if $V' \subseteq V$, $E' \subseteq E$, and $\forall v \in V'$, $l'(v) = l(v)$.*

Definition 2.2. *(Subgraph Isomorphism) Given a query graph $Q(V_q, E_q, l_q)$ and a graph $G(V, E, l)$, a subgraph $G'(V', E', l')$ of G is a subgraph isomorphic match to Q if there exists a bijective function $f : V_q \rightarrow V'$ such that:*

1. $\forall v \in V_q$, $l_q(v) = l'(f(v))$.

2. *An edge (u, v) is in E_q if and only if $(f(u), f(v))$ is in E'.*

The objective of the subgraph isomorphism problem is to find all subgraph isomorphic matches of Q in G. This problem is known to be NP-hard, [25], but with a fixed query size, can be computed in polynomial time with respect to the query size. Looking at Table 2.1,

it is easy to see why the subgraph isomorphism problem is intractable in the worst case. Suppose we have an unlabeled graph with n vertices, and every vertex is connected to every other vertex. Now suppose we use that same graph as a query graph. Then every possible permutation of the vertices in the data graph is subgraph isomorphic to the query graph. This means that there are $n!$ matches, and it would take at least $O(n!)$ time just to enumerate all of them. Luckily, if the query is size n_q, then the number of matches is at most $\frac{n!}{(n-n_q)!}$, which is $O(n^{n_q})$. In practice, because the query graph is often much smaller than the data graph and because the vertex labels reduce the number of possible matches, finding all subgraph isomorphisms can be feasible.

Chapter 3

Techniques for Graph Pattern Matching

As mentioned, one of the current motivations for the development of faster subgraph isomorphism algorithms is for their use in processing queries on graph databases. In [19], the authors present a basic outline for a graph pattern matching algorithm that could be used for this purpose as follows (assuming query graph $Q(V_q, E_q, l_q)$ and vertex ids $0, ..., |V_q| - 1$):

1. Retrieve feasible matches for each vertex in the query graph based on semantic and/or neighborhood information in the data graph. This results in sets $\Phi(0), ..., \Phi(|V_q| - 1)$, where $\Phi(i)$ is the set of feasible matches for vertex $i \in V_q$.

2. (Optional) Reduce the search space globally, refining the set of feasible matches for each query vertex.

3. (Optional) Optimize the search order of vertices in the query.

4. Search the remaining space in a depth-first manner.

This outline describes an entire class of Ullmann-inspired *tree search based* algorithms [32, 11, 30, 36, 35, 19, 9, 16, 2, 25], of which ours is a member. VF2 is also based on tree search

and follows a depth-first strategy as well, but it follows a slightly different approach. The outline serves as a good reference point for several algorithms discussed in the rest of the paper, and we utilize the notation of [19] in describing feasible matches for a vertex i in the query graph as a set $\Phi(i)$. These tree search based algorithms differ mostly in Steps 1-3, and there are various algorithms for each of these [19, 30, 36, 35, 18]. We first discuss Ullmann's algorithm in the context of this outline, before moving to discuss VF2.

3.1 Ullmann's Algorithm

Ullmann's algorithm was the foundational algorithm of the tree search class of algorithms for subgraph isomorphism [9, 16, 25], and so follows the outline above very closely. In Step 1, the algorithm obtains feasible matches $\Phi(u)$ for each vertex $u \in V_q$ by selecting all vertices v in the data graph such that $l(v) = l_q(u)$ and $|adj(v)| \geq |adj(u)|$. In Step 2, the search space is globally pruned by checking that

$$\forall (u, u') \in E_q,\ \forall v \in \Phi(u),\ \exists v' \in \Phi(u') \text{ s.t. } (v, v') \in E \tag{3.1}$$

In other words, every candidate vertex v in the data graph that matches vertex u in the query graph has to have children that match all children of u. It uses a simple iterative algorithm that we describe in the next section to perform this pruning. In order to understand the optimization performed in Step 3, it is best to first understand the search algorithm for Step 4. Given an ordering $u_0, ..., u_{|V_q|-1}$ of query vertices, the search procedure works by beginning with $\Phi(u_0)$ and isolating one of its vertices (call it v). It then performs the pruning procedure as if $\Phi(u_0) = \{v\}$ and recursively calls this search procedure on $\Phi(u_1)$. Meanwhile, the algorithm ensures that no vertex is used to match more than one query vertex. Thus, when the search procedure reaches depth $|V_q|$, each $\Phi(u_i)$ contains a single unique vertex and, because of the conditions of the pruning procedure, an isomorphic match

has necessarily been found. The algorithm then backtracks, until no search path has been left unexplored. Going back to Step 3, Ullmann suggests that the query vertices are processed in order of increasing degree, since this allows more branches to be encountered and pruned early in the process. Though the original algorithm uses adjacency matrices, it can easily be implemented with an adjacency list representation using the same logic.

3.2 VF2 Algorithm

Step 1 of the VF2 algorithm is the same as Step 1 of Ullmann's algorithm. However, VF2 utilizes a "build up" rather than a "prune down" approach, and so Step 2 does not exactly apply to VF2. Rather, search paths are eliminated as an ongoing process during search. From the initial set of feasible matches, VF2 begins by adding a matching *(query vertex, data vertex)* pair to a partial match set, before performing a depth first search through the data graph and the query graph simultaneously starting at these vertices. At each stage it eliminates search paths based on local information. Further details of the algorithm can be found in [11].

3.3 Related Work

The field of subgraph pattern matching has been studied for decades [9]. As previously mentioned, pattern matching differs between two different kinds of graph databases. One consists of a collection of small-medium size graphs (the *graph-transactional* case), whereas the other consists of a single large graph [16]. In the graph transactional setting, the focus of a pattern matching algorithm is to find all graphs in the database that contain a subgraph similar to a query graph (and possibly to return those subgraphs) [16]. In a single graph setting, which we discuss, the goal is to find all subgraphs of a data graph that are similar

to a query graph. Furthermore, the field is divided into exact and inexact matching, which we discussed earlier.

This paper focuses on exact pattern matching in single graph settings. In [25], a very recent survey of the state of the art in subgraph isomorphism, the authors give an experimental comparison between five recent algorithms for subgraph isomorphism in a unified framework: VF2 [11], GraphQL [19], QuickSI [30], SPath [36], and GADDI [35]. They also present several optimization techniques for each of the algorithms. GraphQL [19] follows the outline for tree search based algorithms discussed earlier. It uses neighborhood signatures of data vertices to prune the initial candidate set and a pruning technique called pseudo subgraph isomorphism to globally narrow the search space. Pseudo subgraph isomorphism works by creating a bipartite graph between the query graph and its potential matches in the data graph and iteratively comparing subtrees of greater height, until it reaches a specified depth. This algorithm is designed to work with the single graph database scenario (though it also applies to the transactional setting), but because the storage of the graph is disk-based rather than memory-based, we do not use it in our comparisons.

QuickSI [30] works by preprocessing data graphs to calculate label frequencies and the frequencies of $(fromLabel, edgeLabel, toLabel)$ triples. When a query is requested, it weights the edges of the query graph accordingly and uses these weights to order the search by creating a minimum spanning tree. Despite its demonstrated good performance in [25] on single graph scenarios, they are, however, focused on the graph-transactional setting, attempting to solve the *graph containment* problem, which seeks to find all graphs in the database that contain a query graph [30].

The SPath algorithm [36] is centered around a local path-based indexing technique for vertices in the data graph and transforms a query graph into a set of shortest paths in order to process a query. While it is focused on the single graph scenario, the authors of [25] demonstrate that it is almost always outperformed by GraphQL due to the fact that its

performance is greatly dependent upon the path search order, and its optimization techniques for this order are lacking. And again, it relies on a disk-based indexing technique.

GADDI [35] indexes a data graph based on a new kind of neighborhood distance metric, but [25] shows poor performance compared to the others mentioned. The authors themselves admit that they restrict their applications to small-medium sized graphs [35].

Though in the experimentation in [25], VF2 was outperformed by some of the aforementioned algorithms in several cases, the experimentation for the single graph case was limited to data sets of less than 5,000 vertices and less than 90,000 edges. Furthermore, the algorithms were stopped after 1,000 returned results, which hides the worst-case exponential time complexity of all of the algorithms and makes it difficult to truly tell how the algorithms scale.

Very recently, the same authors of the survey in [25] present an algorithm called $\text{Turbo}_{\text{ISO}}$ which they claim outperforms the others by orders of magnitude in many cases [18]. It introduces the concept of a neighborhood equivalence class for vertices in the query in order to eliminate traversing redundant paths in the search space and uses a novel method for identifying candidate regions of the graph. The paper also indirectly compares $\text{Turbo}_{\text{ISO}}$ to a current state of the art algorithm in distributed graph pattern matching, called STwig [31], by running the algorithm on two of the same datasets. It outperforms the STwig algorithm by up to four orders of magnitude. They also provide runtimes for $\text{Turbo}_{\text{ISO}}$ on synthetic graphs of up to 4 million vertices and 32 million edges, but this is still much smaller than the 10 million vertex, 250 million edge trials demonstrated here.

Chapter 4

Presentation of Algorithm

Our algorithm falls into the category of tree search based algorithms described earlier. The primary difference lies in the pruning algorithm. The graph is stored in memory with adjacency list representation and a separate array that maps vertices to labels. An index is used to map each label to every vertex in G with that label. This allows fast initial access to feasible matches. The memory taken by the graph is thus bounded by the adjacency list size, which is $O(|E|)$.

4.1 Retrieval of Feasible Matches

Referencing the previous outline, our algorithm begins by obtaining feasible matches. For each given query vertex u, it creates $\Phi(u)$ to contain all vertices in the data graph with the same label as u. This is the same as Ullmann's algorithm except that it ignores the degree constraint. On the whole, during experimentation, it turned out that this constraint did not help our algorithm, and in the cases where response times were exceptionally fast, it actually slowed down the algorithm (if only by a few tens of milliseconds). This is because the pruning process rapidly eliminates many of the vertices that would have been eliminated

by an initial degree constraint anyways.

4.2 Pruning Algorithms

4.2.1 Simple Simulation

The pruning procedure used for our subgraph isomorphism algorithm is based on a concept known as graph simulation [20, 26]. The basic version of graph simulation, *simple simulation*, is conceptually equivalent to the refinement procedure used by Ullmann's algorithm for pruning. The conditions in Equation 3.1 can be adapted to create an algorithm as seen in the pseudocode in Figure 4.2 [20]. Because $|\Phi(0)| + ... + |\Phi(|V_q| - 1)| \leq |V||V_q|$, the outer **while** loop in line 3 may execute at most $|V||V_q|$ times. This would correspond to a scenario in which only one vertex is removed with each iteration of the loop. The next two **for** loops (lines 5-6) execute a total of $|E_q|$ times, and there are at most $|V|$ vertices in $|\Phi(u)|$ for any $u \in V_q$, which creates a bound on the innermost **for** loop (line 7). The intersection operation in line 8 takes at most $|V|$ steps, and so the total time complexity is $O(|E_q||V_q||V|^3)$. Though [20] presents an algorithm to reduce the time complexity to be quadratic in $|V|$, it requires a parent list along with an adjacency list to describe all incoming edges into each vertex, which nearly doubles the memory requirement and may be infeasible for large graphs. As demonstrated in Chapter 5, the simple simulation condition alone (Equation 3.1) fails to prune large graphs well enough to be used effectively in obtaining all subgraph isomorphic matches. An example of simple simulation on a graph can be seen in Figure 4.1. Note that vertex 0 is in $\Phi(1)$ despite the fact that it does not have a vertex in $\Phi(0)$ as a parent.

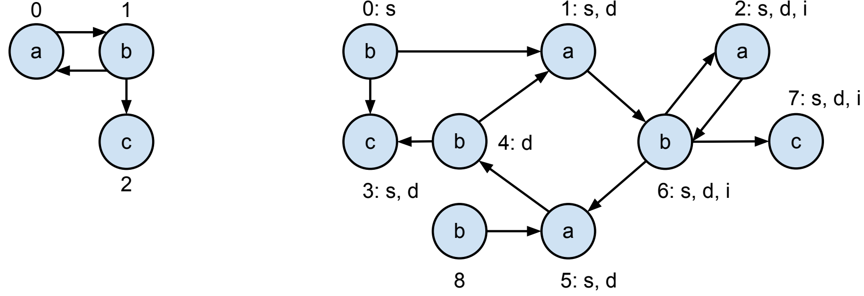


Figure 4.1: An example of a query graph (left) and its matches via simple, dual, and isomorphism in a data graph (right). Simple matches are denoted by an 's', dual matches by a 'd', and isomorphic matches by an 'i'. Note that any vertex that matches via isomorphism also matches via dual, which in turn matches simple.

```
 1: procedure SimpleSim(G, Q, Φ):
 2:     changed ← true
 3:     while changed do
 4:         changed ← false
 5:         for u ← V_q do
 6:             for u' ← Q.adj(u) do
 7:                 for v ← Φ(u) do
 8:                     if G.adj(v) ∩ Φ(u') = ∅ then
 9:                         remove v from Φ(u)
10:                         if Φ(u) = ∅ then
11:                             return empty Φ
12:                         end if
13:                         changed ← true
14:                     end if
15:                 end for
16:             end for
17:         end for
18:     end while
19:     return Φ
20: end procedure
```

Figure 4.2: Pseudocode for the graph simulation algorithm

4.2.2 Dual Simulation

The pruning method used by our algorithm is a straightforward extension of the simple simulation algorithm above, called *dual simulation* [26]. While simple simulation only checks that matches for a query vertex have the appropriate children, dual simulation also checks that they have the appropriate parents. In other words, in addition to the condition in Equation 3.1, matches via dual simulation must also satisfy the condition that

$$\forall (u, u') \in E_q,\ \forall v' \in \Phi(u'),\ \exists v \in \Phi(u) \text{ s.t. } (v, v') \in E \tag{4.1}$$

The pseudocode for this algorithm is shown in Figure 4.3. Unlike the algorithm in [26], it does not require a list of parents for each vertex in the data graph. Notice that the core of the algorithm is the same as the algorithm for simple simulation in Figure 4.2. In the outer two **for** loops (lines 5-6), the algorithm iterates through all pairs $(u, u') \in E_q$. The inner **for** loop (line 8) iterates through all $v \in \Phi(u)$. To check the simple simulation condition, it ensures that the intersection of $adj(v)$ and $\Phi(u')$ is non-empty (line 9). A key observation is that because every vertex in $\Phi(u')$ must contain some parent in $\Phi(u)$, any valid vertex in $\Phi(u')$ must be contained in some such intersection in the course of the iterations of the inner **for** loop (line 8). Thus, we build up a new set $\Phi'(u')$ to contain only those vertices that have a parent in $\Phi(u)$ (line 17), and once all vertices in $\Phi(u)$ have been traversed, we take the new $\Phi(u')$ to be $\Phi(u') \cap \Phi'(u')$ (line 25). In other words, considering $\Phi_v(u')$ to be the children of v that currently match u' (line 9), we have:

$$\Phi'(u') = \bigcup_{v \in \Phi(u)} \Phi_v(u'), \text{ and}$$

$$\Phi(u') = \Phi(u') \cap \Phi'(u')$$

This reduces runtime by filtering on both conditions simultaneously and allows the operation of the algorithm without an explicit list of parents. Though it might seem like dual simulation would take longer than simple simulation to run, in practice it is often faster due to the fact that it prunes the search space much more rapidly. The worst-case time complexity is the same as that of our implementation of simple simulation.

To view the distinction between simple simulation and dual simulation, revisit the example in Figure 4.1. In terms of the query graph structure, we have that $E_q = \{(0, 1), (1, 0), (1, 2)\}$. This means that the vertices in Φ must satisfy the following conditions:

1. Every vertex in $\Phi(0)$ must have both a child and a parent in $\Phi(1)$.

2. Every vertex in $\Phi(1)$ must have both a child and a parent in $\Phi(0)$ and a child in $\Phi(2)$.

3. Every vertex in $\Phi(2)$ must have a parent in $\Phi(1)$.

In contrast with simple simulation, vertex 0 in the data graph is eliminated from $\Phi(1)$ because it does not have a parent in $\Phi(0)$. The four vertex cycle {1, 6, 5, 4} serves as a match to the {0, 1} cycle in the query graph, because, although there is not a one-to-one correspondence between them, each vertex in $\Phi(0)$ has a child in $\Phi(1)$, and vice versa; and every vertex in $\Phi(1)$ has a child in $\Phi(2)$. This preserves both the parent and child relationships from the query *in terms of their feasible matches*. This demonstrates one scenario where the results of dual simulation can contain vertices that are not in any isomorphic match. The other situation where this can occur is when one vertex in the data graph maps to more than one vertex in the query graph. Although these scenarios are possible, as demonstrated later, dual simulation often eliminates most vertices not found in isomorphic matches. This is the key reason behind the speed of our dual-based isomorphism algorithm. Furthermore, these examples are often pruned quickly in the isomorphism algorithm, as will be seen in the next section.

```
1: procedure DualSim(G, Q, Φ):
2:     changed ← true
3:     while changed do
4:         changed ← false
5:         for u ← V_q do
6:             for u' ← Q.adj(u) do
7:                 Φ'(u') ← ∅
8:                 for v ← Φ(u) do
9:                     Φ_v(u') ← G.adj(v) ∩ Φ(u')
10:                    if Φ_v(u') = ∅ then
11:                        remove v from Φ(u)
12:                        if Φ(u) = ∅ then
13:                            return empty Φ
14:                        end if
15:                        changed ← true
16:                    end if
17:                    Φ'(u') ← Φ'(u') ∪ Φ_v(u')
18:                end for
19:                if Φ'(u') = ∅ then
20:                    return empty Φ
21:                end if
22:                if Φ'(u') is smaller than Φ(u') then
23:                    changed ← true
24:                end if
25:                Φ(u') = Φ(u') ∩ Φ'(u')
26:            end for
27:        end for
28:    end while
29:    return Φ
30: end procedure
```

Figure 4.3: Pseudocode for the dual simulation algorithm

4.3 Dual-based Isomorphism

Because dual simulation prunes the search space so effectively, it makes subgraph isomorphism queries possible even on large graphs. The pseudocode for the algorithm can be seen in Figure 4.4. It starts by finding feasible matches for each vertex u in the query graph by retrieving all vertices v in the data graph such that $l(v) = l_q(u)$. This step uses an index that maps labels to vertices in the data graph. It is worth noting that the procedure to compute feasible matches could utilize any necessary vertex attributes or other information available, and the remaining isomorphism algorithm would remain unchanged. This means that an extension of the algorithm to include multiple vertex attributes would be straightforward.

Following this step, dual simulation is used to prune the global search space. This often eliminates most of the extraneous vertices from the match sets. Next, a procedure is invoked to traverse the remaining matches in a depth-first manner. This works the same way as in Ullmann's algorithm. First, a copy Φ' is made of Φ (line 14), and a vertex v in $\Phi(0)$ is isolated and treated as if it were the only vertex to match query vertex 0 (line 15). In other words, $\Phi'(0)$ becomes equal to $\{v\}$. Dual simulation is then performed on Φ', which necessarily removes all vertices in $\Phi(1), ..., \Phi(|V_q|-1)$ that are not contained in an isomorphic match with $f(0) = v$ (assuming f is the bijection defining the isomorphism as described in Definition 2.2). If dual simulation eliminates all vertices in Φ' at this point, then no isomorphic match exists with the current mapping, and the algorithm backtracks. Otherwise, the search procedure is then called recursively, incrementing the depth until maximum depth is reached or until dual simulation eliminates all remaining possible matches. We experimented with several simple order selection schemes, but none showed any average improvement over simple sequential traversal. Exploring more sophisticated order optimization techniques could be a topic of future work.

To picture the operation of the algorithm, consider again the example in Figure 4.1. The

first step retrieves all feasible matches based on labels, which returns

$$\Phi(0) = \{1, 2, 5\},$$

$$\Phi(1) = \{0, 4, 6, 8\}, \text{ and}$$

$$\Phi(2) = \{3, 7\}.$$

Following this, dual simulation is performed on Φ, yielding the vertices marked with 'd' in Figure 4.1. This eliminates nodes 0 and 8 from $\Phi(1)$. Next, the search process begins by creating a version Φ' of Φ such that $\Phi'(0) = 1$. Thus, when dual simulation is performed again on Φ', vertex 4 loses its necessary parent (vertex 5) and vertex 6 loses its necessary child (vertex 2). Consequently, these vertices are removed from $\Phi'(1)$, which becomes empty, and so dual simulation returns no result, causing the algorithm to backtrack. When the same process is undergone with $\Phi'(0) = \{2\}$, however, dual simulation only removes vertex 4 from $\Phi'(1)$ and vertex 3 from $\Phi'(2)$, leaving $\Phi'(0) = \{2\}$, $\Phi'(1) = \{6\}$, and $\Phi'(2) = \{7\}$, which contains only the vertices found in the isomorphic mapping. It recurses with $depth = 1$, finds that all vertices still satisfy dual simulation, recurses again until $depth = 3$ and then returns a match. It backtracks up to the top level, and when vertex 5 is isolated in $\Phi'(0)$, dual simulation returns no result. The algorithm completes having found the only isomorphic match.

4.3.1 Proof of Correctness

The correctness of the algorithm is now demonstrated.

Lemma 4.1. *The initial Φ contains all possible subgraph isomorphic matches for each vertex in the query graph, and furthermore, no subgraph isomorphic matches are eliminated in the process of search.*

Proof. The initial retrieval of feasible matches is based purely on labels, and all vertices are included that satisfy the label condition of subgraph isomorphism for each query vertex. Thus, no valid vertices are precluded in this way. If a subgraph of the data graph is isomorphic to the query graph, then condition 2 of Definition 2.2 holds for the vertices and edges in that subgraph. This implies that the dual simulation conditions are also satisfied for these vertices, and therefore, dual simulation never eliminates a vertex that occurs in a subgraph isomorphic match. □

Lemma 4.2. *All results of the dual-based isomorphism algorithm are subgraph isomorphic to the query graph.*

Proof. In the first case, when the dual-based isomorphism algorithm returns no matches, by Lemma 4.1, there are no subgraph isomorphic matches in the data graph.

In the other case, when maximum depth is reached, if Φ is non-empty, the following conditions hold:

1. Each $\Phi(i)$ contains a single unique vertex. Thus, Φ is a bijection from the query graph onto a subset of the vertices of the data graph. Call this subset V'.

2. For each vertex u in the query graph, $l_q(u) = l(\Phi(u))$ (due to the condition on the initial retrieval of feasible matches).

3. The condition in Equation 3.1 holds. Because $\Phi(u)$ and $\Phi(u')$ are now unique vertices, rather than sets, this condenses to:

$$\forall (u, u') \in E_q,\ (\Phi(u), \Phi(u')) \in E.$$

   Call the set of all such edges in the data graph E'. Then we have that $(u, u') \in E_q$ if and only if $(\Phi(u), \Phi(u')) \in E'$.

Letting l' be the projection of l onto V', by Definition 2.2, the subgraph $G'(V', E', l')$ is necessarily an isomorphic match to Q with bijection $\Phi : V_q \rightarrow V'$. □

Lemma 4.3. *The algorithm always successfully terminates.*

Proof. See runtime analysis in Section 4.3.1. □

Theorem 4.1. *The algorithm for dual-based isomorphism correctly finds all subgraph isomorphic matches of a query graph in a data graph.*

Proof. This follows directly from the previous lemmas. □

Runtime Analysis

Following the logic in Section 2.2.2, the worst-case number of matches given a query graph $Q(V_q, E_q, l_q)$ a data graph $G(V, E, l)$ is $O(|V|^{|V_q|})$. Dual simulation must then be executed on each of these O($|V_q|$) times (the maximum depth of the search tree) in the worst case, yielding a time complexity of $O(|V|^{(|V_q|+3)}|V_q|^2|E_q|)$.

```
1: procedure FindMatches(G, Q)
2:     matches ← ∅
3:     Φ_0 ←FeasibleMatches(G, Q)
4:     Φ_0 ←DualSim(G, Q, Φ_0)
5:     Search(G, Q, Φ_0, 0)
6:
7:     procedure Search(G, Q, Φ, depth)
8:         if depth = Q.size then
9:             matches ← matches ∪ Φ
10:        else
11:            for v ← Φ(depth) do
12:                if v ∉ Φ(0) ∪ ... ∪ Φ(depth − 1)  then
13:                    Φ′ ← copy of Φ
14:                    Φ′(depth) ← {v}
15:                    Φ′ ←DualSim(G, Q, Φ′)
16:                    if Φ′ is not empty then
17:                        Search(G, Q, Φ′, depth + 1)
18:                    end if
19:                end if
20:            end for
21:        end if
22:    end procedure
23:
24:    return matches
25: end procedure
```

Figure 4.4: Pseudocode for the dual-based isomorphism algorithm

Chapter 5

Experimentation

In this section, we seek to identify the impact of various factors on the runtime of the dual-based isomorphism algorithm, and to display its performance on two real life datasets. The factors explored are the number of distinct labels in the data graph, the number of vertices in the query, the number of vertices in the data graph, the degree distribution in the data graph, and the density of the data graph. The real datasets are geared more towards a demonstration of feasibility for our algorithm on real life problems. All experiments were run on a machine with two 2GHz Intel Xeon E5-2620 CPUs, each having six cores, and 128G of DDR3 RAM. Both Ullmann's algorithm and the dual-based isomorphism algorithm were written in Scala version 2.10. The Ullmann's algorithm implementation used an adjacency list representation rather than an adjacency matrix and had access to the same data structures as the dual-based isomorphism algorithm. For the VF2 algorithm, the implementation provided by the iGraph library [12] was used due to its dedication to high performance and its ability to scale to large graphs.[1] This implementation was written in C.[2]

Two methods were used for the generation of synthetic graphs. The first graph generator

[1] In contrast, the original vflib [11] package has a limit of 65535 vertices per graph.

[2] According to http://benchmarksgame.alioth.debian.org/, across ten benchmarks C was 2X faster than Scala at the median, at best 6X faster, and at its worst, still slightly faster.

created random graphs by connecting each vertex to a random number of other vertices based on a given desired average degree. Like [26] and [27], we specify a variable α to describe the relationship between the number of edges and the number of vertices in the graph, such that $|E| = |V|^{\alpha}$. In other words, the average degree is equal to $|V|^{\alpha-1}$. Unless specified, we use $\alpha = 1.2$ to coincide with the metric used by others [26, 27]. The other generator created graphs with power law degree distributions, as can be found in many real world graphs [2, 15, 17, 3, 23, 24, 5, 4, 28, 1]. Power law graphs contain a small number of "hubs" with a very high degree and many vertices with a much smaller degree. It takes as input a maximum degree and an exponent, and randomly chooses degrees for each vertex with a power law distribution from 0 to that maximum. We chose an exponent of 2.1 for our experiments, which was the reported exponent of the Internet AS graph [2, 15]. Here the maximum degree was chosen to preserve the same average degree according to the parameter α.

Queries were also generated in two different ways. One generator worked by choosing a random vertex in the data graph and performing a Breadth-First Search (BFS) until it obtained the desired number of vertices. This ensured that there was at least one subgraph isomorphic match in the data graph. The other method simply worked by generating a small random graph of the specified size. This has the benefit of exposing the system to more unpredictable behavior. In both cases, the average degree of queries was chosen such that $\alpha = 1.2$. It was ensured that all query graphs were connected graphs, because using a disconnected query graph is equivalent to running multiple queries. When not specified, the queries were generated using the BFS technique.

Note that we do not stop experiments after a limited number of results have been returned, but allow them to run their full course. This gives a better picture of the way the algorithm truly behaves on different inputs.

5.1 Synthesized Data

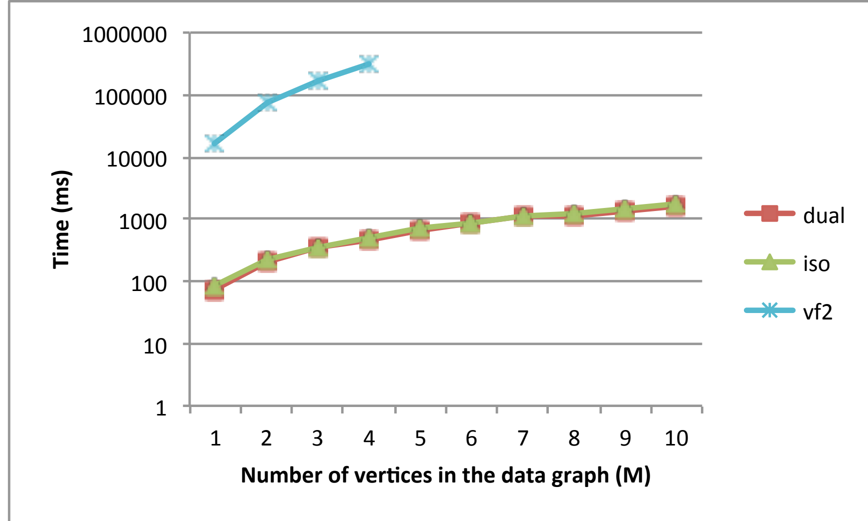
This portion of the experimentation was dedicated to determining the effects of various factors on the runtime of our algorithm, Ullmann's algorithm, and the VF2 algorithm.

5.1.1 Effect of Data Graph Size

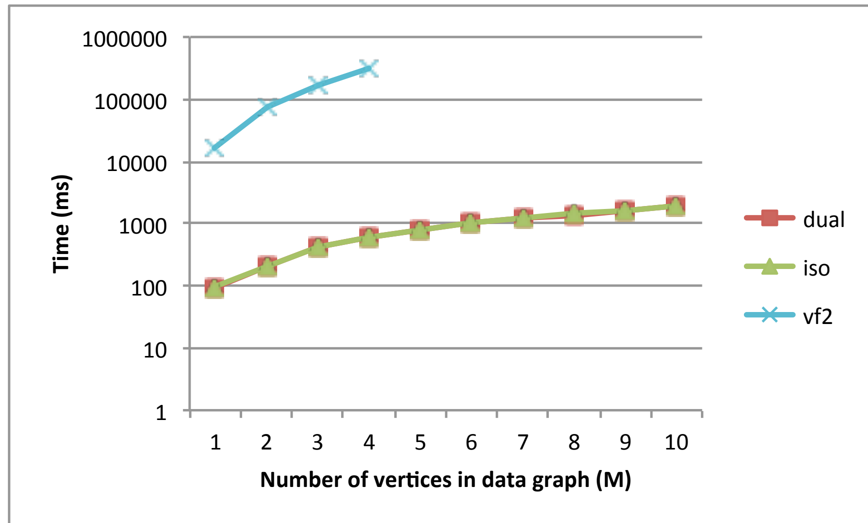
The effect of the size of the data graph on the runtime is important for scalability. Looking at Figure 5.1, it can be seen that our algorithm scales well with graph size given a constant number of unique labels (in this case, 100) and a constant query size. In the experiments shown, it is up to 450 times faster than VF2. The separation continued to grow, and it became infeasible to run VF2 after 3,000,000 vertices. The largest of these graphs had 10 million vertices and over 250 million edges. The behavior of the runtime is very similar for random graphs and for power law graphs. This demonstrates the robustness of the algorithm to various structural types. To the best of our knowledge, this is the largest of any graphs to be tested on a centralized subgraph isomorphism algorithm in the literature.

5.1.2 Effect of query size

Because the query size $|V_q|$ is present in the exponent of the time complexity, it makes sense for runtime to increase as the query size increases. Looking at Figure 5.2, this is the case. Note again, however, that the number of matches is not limited in these experiments. The average number of matches returned in the queries in Figure 5.2d was 443; for queries of size 100, it was 1,599. It makes sense that the runtime increases as the number of matches increases, because it takes longer to enumerate them all. In Figure 5.2a, there is a large spike in the runtime of Ullmann's algorithm. This is due to a single case where the algorithm timed out after 30 minutes. In this case, there were 0 isomorphic matches, but simple simulation returned 51,696 unique candidate vertices, with 10,186 candidates for a single node, whereas



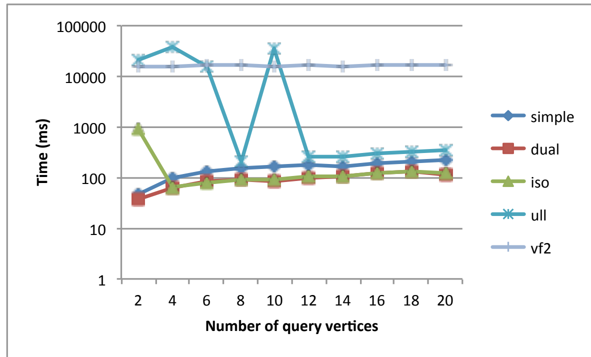
(a) Random Degree Distribution, $|V_q| = 10$



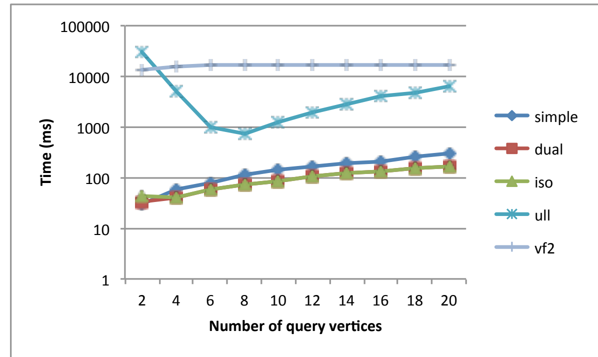
(b) Power Law Degree Distribution, $|V_q| = 10$

Figure 5.1: The average runtimes of 10 queries for each graph size indicated on 5 of each kind of graph.

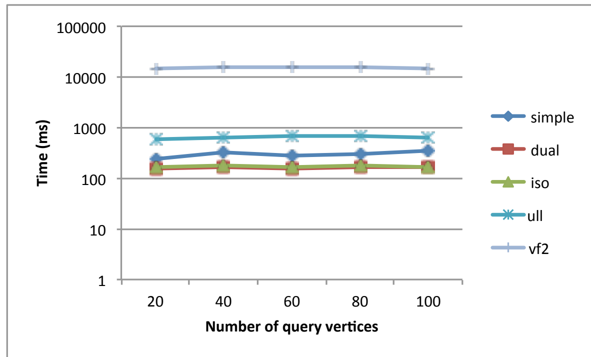
dual simulation returned 0 candidate vertices. This displays the huge difference that an effective pruning algorithm can make. In another case, all four isomorphism algorithms timed out after 30 minutes, and so this outlier was omitted.



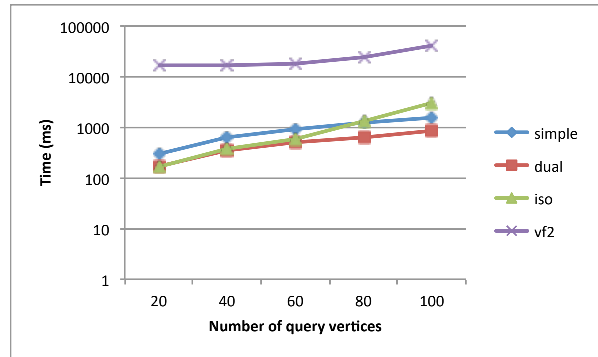
(a) Small query sizes, random queries, $|V| = 10^6$



(b) Small query sizes, BFS queries, $|V| = 10^6$



(c) Large query sizes, random queries, $|V| = 10^6$



(d) Large query sizes, BFS queries, $|V| = 10^6$

Figure 5.2: The average runtimes of 10 queries for each query size indicated on each of 5 different randomly generated graphs. Ullmann's algorithm is not displayed in the large BFS queries because it took a prohibitively long time in many cases.

5.1.3 Effect of Labels

Because the number of unique labels greatly affects the number of feasible matches for each vertex in a query, it should be expected to have a tremendous impact on running time. As can be seen in Figure 5.3, this is the case for all of the algorithms tested. For these experiments, we generated graphs with the number of specified unique labels randomly distributed across the vertices. This means that for a graph with n labels, the filtering power of a label is approximately $\frac{1}{n}$. Therefore, the more labels there are, the fewer vertices there are in the search space before initial dual simulation. This leads to fewer paths traversed and fewer

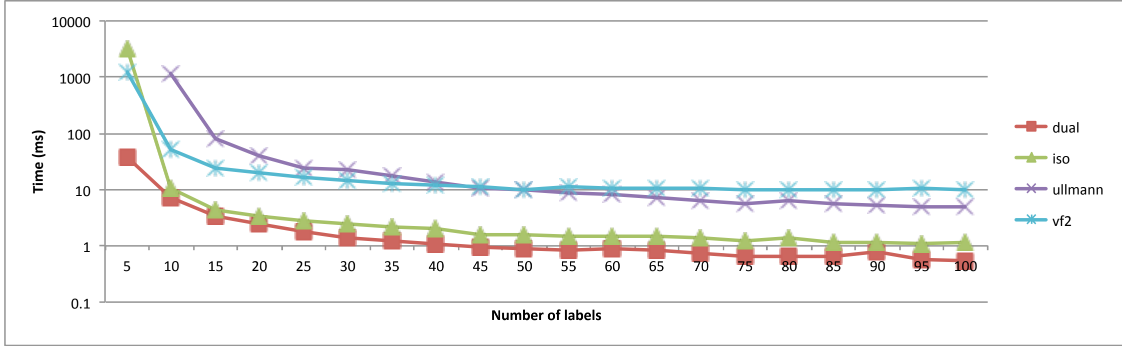


Figure 5.3: The average runtimes of 10 queries for each number of labels indicated for each of 5 different randomly generated graphs, $|V| = 10^4$.

matches. It makes sense that graph pattern matching would take the longest in the case of an unlabeled graph (one with only one label) because every vertex in the data graph is initially a feasible match for every vertex in the query graph. Note that VF2 performs better than our algorithm in graphs with only five labels in Figure 5.3. This is because, without the pruning power of labels, the Ullmann-style prune-down approach is less effective than the VF2 style which takes a build-up approach from candidate matches. As the number of labels increases, our algorithm begins to outperform VF2.

5.1.4 Effect of Data Graph Density

The next synthetic experiments deal with the effect of graph density on query response time. The number of vertices in the data graph was fixed at 10^6, and the number of labels was fixed at 100. Looking at Figure 5.4, the runtime increases as the density increases, with a turn upwards when the average degree reaches 70. The overall growth is due to an increase in matches brought on by the increase in density. The reason it begins to grow so sharply with an average degree above 70 is that the degree is approaching the number of labels, which means that the probability of a given data vertex satisfying the conditions of dual simulation

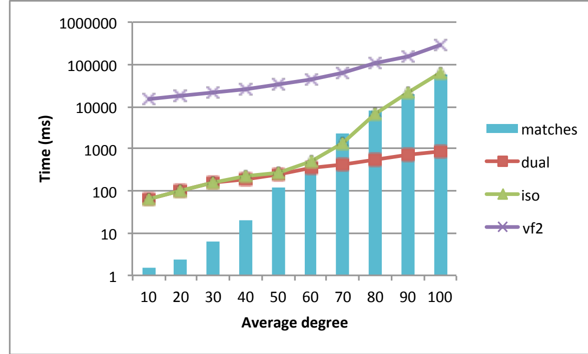


Figure 5.4: The effect of graph density on runtime.

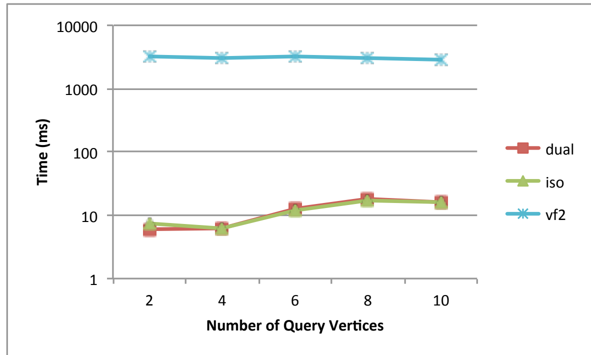
increases greatly. This causes the initial size of Φ to grow, which in turn increases the number of possible matches at the start of search, as well as the number of actual matches to be found. For example, the average number of matches for graphs with an average degree of 100 was 55,426 compared to 18,872 for graphs with an average degree of 90, all the way down to an average of 1.45 matches for average degree 10. It can be seen in Figure 5.4 that the runtime very closely follows the number of matches after the initial pruning time via dual simulation. The takeaway here is that, though the runtime is increasing, the output is increasing proportionally as well. In a practical scenario, limiting the number of results or processing the results as they are returned could ameliorate this issue.

5.2 Real Data

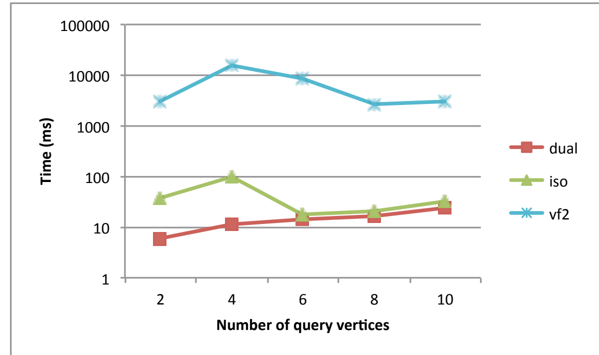
5.2.1 Youtube Dataset

The Youtube dataset[3] is a graph consisting of 509,962 vertices and 2,930,310 edges, where each vertex represents a unique video and a directed edge from one vertex to another indicates that they are related. We performed 100 queries for each size shown in Figure 5.5. Both

[3]http://netsg.cs.sfu.ca/youtubedata/



(a) BFS Queries



(b) Random Queries

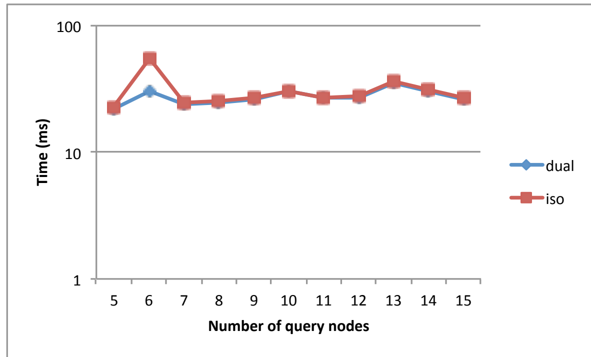
Figure 5.5: The average runtimes of 100 queries of each size on the Youtube dataset.

randomly generated and BFS queries were used. The graph was given 100 randomly distributed labels. In both cases, runtime increased slowly as query size increased, as would be expected. The random tests show a spike in the queries with $|V_q| = 4$ because of one query with 25,398 matches that took 6.94 seconds to run. Another query with 88,350 matches took approximately 1 second. In the case of the BFS queries, runtimes are reasonably stable across query sizes. In all cases, the queries took less than 100 ms on average.

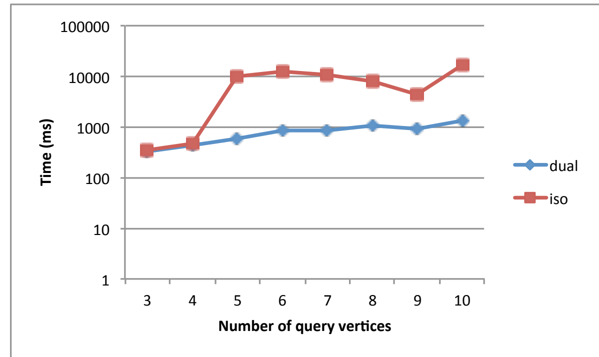
5.2.2 Wordnet Dataset

The WordNet dataset[4] uses English words as vertices and various relationships between them as edges. For example, if two words are synonyms or if one word is a subtype of another, they are linked. Vertices are labeled with one of five parts of speech. There are 82,670 vertices and 133,445 edges in the dataset. We perform 100 queries of each type for each of the query sizes specified. To compare to TurboISO and STwig [18, 31], we stop our algorithm after the first 1024 results. While [18] and [31] both use depth-first search queries, the experiments here were run using the BFS queries described earlier because they yielded a higher average

[4]http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.zip



(a) Random Queries



(b) BFS Queries

Figure 5.6: The average runtimes of 100 queries of each size on the WordNet dataset. The times are shown only for the first 1024 results, if necessary.

degree and we felt that they more realistically describe practical queries. Note that the average degree is 2 for all of these queries.

Looking at the random queries in Figure 5.6a, the runtimes are stable across query sizes. It is important to note, however, that no results were returned in any query with above 7 vertices, and on average, there were only 2 matches. In contrast, the average number of results returned in the BFS queries in Figure 5.6b is 993. Though we may not directly compare to $\text{Turbo}_{\text{ISO}}$ or STwig, future work could include a direct comparison between them.

5.3 Effectiveness of Dual Simulation

The key behind the dual-based isomorphism algorithm is the effectiveness of dual simulation in pruning unnecessary vertices. In over 5,000 queries tested, dual simulation returned only 4.1% more unique vertices than subgraph isomorphism. In 96.6% of those cases tested, it returned only those vertices that were contained in some subgraph isomorphic match. This is in contrast to simple simulation, which on average returns 5,339 times the number of unique vertices contained in subgraph isomorphic matches, and it only gave exactly the

same vertices in 10.2% of cases. This is because, as explained earlier, there are only a few select situations where a node can be part of a dual simulation match and not part of an isomorphic match. The parent and child restrictions, though seemingly naive, are incredibly effective. This explains why, in so many cases, dual-based isomorphism barely takes longer than dual simulation. This is not to say that it is perfect; in the 3.4% of cases that it differs from subgraph isomorphism, it returns 2.2 times the number of unique vertices on average. On the whole, dual simulation serves as an incredibly effective and efficient way to prune the search space.

Chapter 6

Conclusion & Future Work

In this paper, we have presented a novel algorithm for subgraph isomorphism that is superior to or at least competitive with state of the art algorithms in many cases on large labeled graphs. We present a simple pruning technique, based on dual simulation, that is shown to drastically reduce the search space, often eliminating all vertices not contained in some subgraph isomorphic match. We have demonstrated this with extensive experimentation on both synthetic and real graphs. We covered the major factors that influence the runtimes of the algorithms, and we explained why, and in what cases, exact matching can be practical for query processing on large graphs. Scala code for the dual-based isomorphism algorithm can be found in the graphalytics package of ScalaTion.[1] The algorithm is relatively simple and required 21 only lines of code each for dual simulation and dual-based isomorphism.

Future work could include extending the algorithm to work with multiple vertex attributes, multiple edges and edge attributes, and wildcard queries. Because dual simulation is often the bottleneck of the algorithm, ways to parallelize or speed up the dual simulation algorithm would bring further improvements. Adapting the algorithm to work on graphs stored on disk or in a distributed environment would allow it to handle even larger graphs.

[1] `http://www.cs.uga.edu/~jam/scalation`. See Appendix A for more details.

We would also like to compare ours with the Turbo$_{\text{ISO}}$ algorithm on larger graphs. Overall, the utilization of dual simulation in a subgraph isomorphism algorithm leads to rapid pruning of the search space, which, given a sufficient number of labels, allows for fast query processing independent of graph size.

Bibliography

[1] Lada A Adamic and Bernardo A Huberman. The web's hidden order. *Communications of the ACM*, 44(9):55–60, 2001.

[2] Charu C Aggarwal and Haixun Wang. *Managing and mining graph data*, volume 40. Springer, 2010.

[3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Structural properties of the African web. In *The Eleventh International WWW Conference*, volume 66, 2002.

[5] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.

[6] Horst Bunke and G Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[7] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 199–208. ACM, 2009.

[8] William J. Christmas, Josef Kittler, and Maria Petrou. Structural matching in computer vision using probabilistic relaxation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):749–764, 1995.

[9] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[10] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *CoRR*, cs.AI/9402102, 1994.

[11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[12] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5), 2006.

[13] Luc Dehaspe, Hannu Toivonen, and Ross D King. Finding frequent substructures in chemical compounds. In *KDD*, volume 98, page 1998, 1998.

[14] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. Frequent substructure-based approaches for classifying chemical compounds. *Knowledge and Data Engineering, IEEE Transactions on*, 17(8):1036–1050, 2005.

[15] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[16] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53, 2006.

[17] Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1371–1380. IEEE, 2000.

[18] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 international conference on Management of data*, pages 337–348. ACM, 2013.

[19] Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.

[20] Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. Computing simulations on finite and infinite graphs. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 453–462. IEEE, 1995.

[21] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. The web as a graph: Measurements, models, and methods. In *Computing and combinatorics*, pages 1–17. Springer, 1999.

[22] Stefan Kramer, Luc De Raedt, and Christoph Helma. Molecular feature mining in hiv data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 136–143. ACM, 2001.

[23] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D Sivakumar, Andrew Tomkins, and Eli Upfal. Stochastic models for the web graph. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 57–65. IEEE, 2000.

[24] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Extracting large-scale knowledge bases from the web. In *VLDB*, volume 99, pages 639–650. Citeseer, 1999.

[25] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 133–144. VLDB Endowment, 2012.

[26] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *Proceedings of the VLDB Endowment*, 5(4):310–321, 2011.

[27] M Usman Nisar, Arash Fard, and John A Miller. Techniques for graph analytics on big data. In *Proceedings of the IEEE International Congress on Big Data*, pages 255–262, 2013.

[28] Sidney Redner. How popular is your paper? an empirical study of the citation distribution. *The European Physical Journal B-Condensed Matter and Complex Systems*, 4(2):131–134, 1998.

[29] Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, (3):353–362, 1983.

[30] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[31] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[32] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[33] Shinji Umeyama. An eigendecomposition approach to weighted graph matching problems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 10(5):695–703, 1988.

[34] Stanley Wasserman. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[35] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203. ACM, 2009.

[36] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.

Appendix A

Instructions for Download and Execution of Source Code

The code used in the paper is available at `http://www.cs.uga.edu/~jam/scalation` in the graphalytics package. After downloading ScalaTion as described in the README section, tests for simple simulation, dual simulation, an adjacency list version of Ullmann's algorithm, and the dual-based isomorphism algorithm can be run by executing the command

```
scala -cp classes scalation.graphalytics.PatternMatcherTest
```

The code for the test can be found in the file `PatternMatcherTest.scala`. The graphalytics package is independent of other packages, with the exception of the files `ColorDag.scala`, `ColorTree.scala`, and `ShortestPath.scala`. If these files are removed, the graphalytics package can be successfully compiled on its own. The current version of ScalaTion, version 1.0, uses Scala 2.9, but ScalaTion 1.1 uses Scala 2.10.