Towards Discovery of Hidden Adversarial Networks in Large Database Graphs

by

John Schwartz

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the degree of

MASTER OF SCIENCE

Major Subject: Computer Science

Approved:

____

Mark Goldberg, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

December, 2010

© Copyright 2010
by
John Schwartz
All Rights Reserved

CONTENTS

Towards Discovery of Hidden Adversarial Networks in Large Database Graphs i

LIST OF FIGURES v

ACKNOWLEDGMENT vi

ABSTRACT vii

1. INTRODUCTION 1

 1.1 Problem Description 1

 1.2 Background 1

 1.3 Previous Work 2

 1.4 Two Approaches 3

 1.5 Definitions and Assumptions 3

2. FREQUENCY VECTOR 5

 2.1 Definition 5

 2.2 Indexing Step 6

  2.2.1 Kd-tree Details 6

 2.3 Selection of Representative Node 6

  2.3.1 Populous Method 7

  2.3.2 Diverse Method 7

 2.4 Search Step 7

 2.5 Scoring Step 8

 2.6 Analysis 9

3. TYPE MATRIX 11

 3.1 Indexing Step 11

 3.2 Search Step 12

 3.3 Analysis 13

4. Generation 14

 4.1 Random Graphs 14

4.2 Wikipedia Database Graph 14

5. Results 16

5.1 Random Graph Results 16

5.2 Wikipedia Graph Results 25

6. Conclusions 28

7. FUTURE WORK 29

LITERATURE CITED 30

LIST OF FIGURES

Figure 2.1: Frequency-vector diagram. 5
Figure 5.1: Cumulative success as measured as a factor of recall. Size 10 fragments... 16
Figure 5.2: Cumulative as measured as a factor of recall. Size 20 fragments. 17
Figure 5.3: Accuracy of frequency-vector across several parameters. Size 10 fragment. Lines are average degree of each node. 18
Figure 5.4: Frequency-vector algorithm on randomly constructed graphs...................... 18
Figure 5.5: Accuracy of frequency-vector across several parameters. Size 30 fragment.19
Figure 5.6: Accuracy of type-matrix on size 10 fragments. 20
Figure 5.7: Type-Matrix approach. Compare to Figure 5.4. 20
Figure 5.8: Accuracy of type-matrix on size 30 fragments. 21
Figure 5.9: Average time for a variety of tests. 22
Figure 5.10: Average time for a variety of tests. 22
Figure 5.11: Average percent overlap of size 30 fragments. 23
Figure 5.12: Average percent overlap of size 30 fragments. 23
Figure 5.13: Distribution of overlap across 1000 tests using type-matrix...................... 24
Figure 5.14: Frequency-vector method running on three Wikipedia databases, each with a different number of types. The lines represent 10, 20, and 30 fragment size tests. 25
Figure 5.15: Type-matrix method running on three Wikipedia databases. Lines represent fragment size. 25

ACKNOWLEDGMENT

I would like to thank Dr. Mark Goldberg for his guidance and ready nature. This project has come a long way thanks to his advice and direction. I'd also like to extend thanks to Dr. Malik Magdon-Ismail and Dr. William A. Wallace for their comments and criticisms that helped shape the development of this project; the faculty and staff of the RPI CS department for all their work towards maintaining the environment for research to thrive on campus; and of course Mr. Josh Greenman for being a pleasure to work with for the duration.

ABSTRACT

This thesis describes the framework of a comprehensive system for locating hidden adversarial networks within a large database, based on partial information. Currently, no systematic methods have been developed for the problem of discovering potential matches to fragments in this situation specifically. We present two distinct algorithms, each with their own advantages and disadvantages. The algorithms are presented in three steps: first an indexing step which can be performed offline and infrequently, second a search step which is performed at runtime, and finally a scoring step to return the best results to the user. Our system is shown to work very well for certain graph structures.

1. INTRODUCTION

1.1 Problem Description

An intelligence analyst has developed a semantically labeled graph containing labeled nodes and links which represent the results of an investigation of a hidden adversarial network connecting people, places, events etc. Suppose the analyst wishes to compare the data collected so far, *the fragment*, with the data in a central database which contains the union of previously collected investigations.

In this example the information collected by the analyst may be scarce and incomplete, while the information in the database accumulated over a long period of time would be more complete. Conversely, the newly acquired data may contain information not yet discovered, or unavailable during previous investigations. The objective from the analyst's point of view is to discover, in the central database, statistically significant approximate matches to the fragment. This allows the analyst to update the appropriate data in the database, or to get a more complete understanding about the fragment and it's interactions with the rest of the data.

1.2 Background

We can model this problem using semantic graphs. Each object within the system (the people, places, events etc) can be modeled as a node and their relations to each other can be modeled with links, or *edges*. The problem that we have informally described above thus relates to the classical computational problem of identifying a subgraph, within a given large graph, which is isomorphic to a given fragment graph [1]. The subgraph isomorphism problem is a more generic form of the graph isomorphism problem. Essentially, the graph isomorphism problem seeks to determine whether a query graph Q is the same as, or *isomorphic* to, a graph G. The subgraph isomorphism problem seeks to determine if Q is isomorphic to any *subgraph* of G.

A naive approach to this problem is to simply sequentially investigate each possible subgraph of G and check whether each one is isomorphic to the fragment. This approach is clearly inefficient due to the large number of possible subgraphs, which is related to the fact that this problem in its theoretically setting is known to be NP-hard. This

suggests that finding the exact solution will be too slow for practical use. Algorithms can be designed which will be faster, but generally sacrifice guaranteed correctness for an improvement in speed. This represents only one part of the challenge of this problem. The other part of the challenge is that at the root of our efforts is our desire to produce a system which will be useful to a human analyst. The need to incorporate human specialists, with knowledge and intuition of the investigation, requires that we need to produce an algorithm which attempts to bring the most statistically useful information to the users attention. This is because no matter how sophisticated the software package may be, it cannot fully replace human experience. A human's tolerance for searching through data is limited, so we must be careful not to provide too much for the analyst to examine. Thus we attempted to design our work with these factors in mind, to provide a fast algorithm which will produce a limited number of confident results that a human analyst could then examine.

1.3 Previous Work

The problem of subgraph isomorphism is not a new one [2]. There have been several incarnations of the problem, with various different restrictions. A problem which is similar to ours is that of querying a database of graphs which have a size which is of a similar order of magnitude as the query. The traditional approach is to perform the search via graph-based indices [3]. By indexing the graphs before hand, the search step can be done much quicker. Indexing graphs based upon edges or connected components can make difficult problems much simpler [4].

Currently, no systematic methods have been developed for the problem of discovering potential matches to fragments in a single large database graph. At present there are several methods of identifying clusters of related topics, but no principled automated approaches to identifying either individual or group manipulative behavior in Wikipedia or similar entities.

1.4 Two Approaches

In order to speed up search time, we apply the general concept of indexing to our problem of searching for a sub-structure, *the fragment*, in the large structure, *the database semantic graph*. Thus, the problem is split into two stages:

1) An offline indexing of the large structure; and

2) An online search which employs the results of indexing.

In this situation, offline refers to running a process which the user does not need the results of immediately, while online refers to running a process which the user will be patiently waiting for the results of. In the practical setting, indexing would be performed infrequently since the database would not change often, so it could be run offline, such as overnight. The search which uses the index would be executed for each such request and would thus be expected to run quickly, since an analyst will expect the results quickly, such as one might expect of an internet search.

Our approaches take advantage of properties of the graphs which would appear in the practical setting. We assume that both the *database graph* and the *fragment graph* are semantic graphs whose nodes are assigned special labels, or *types*. The starting point for both approaches to *(1)* is the utilization of the existence of discernable node types in the semantic graph. These types could be *person*, *location*, *event*, *date*, etc. The important difference between types and general labels is that types cannot be *confused*. That is, a person inherently cannot be confused with a location, an event, or a date. Thus any mapping proposing to map a person to a location is invalid.

As an extension of this idea, we can assign each node a vector, whose coordinates are the based on number of links connecting the node to its neighbors of each given type. We present two systems of algorithms, *frequency-vector* and *type-matrix*, which will build upon this idea to index the database in a manner such that search will be reasonably fast and accurate.

1.5 Definitions and Assumptions

The research presented in this paper is attempting to allow a human analyst to locate hidden adversarial networks within a database based upon a graph provided by him/her. In this situation, a network refers to a collection of information connected in various

meaningful ways. The network is hidden because the analyst does not know the location of the network within the database when they begin the search. The network is considered adversarial because the information is not guaranteed to be correct or complete.

Throughout this paper we will discuss a *database graph G*, a *fragment graph F*, and one or more *subgraphs* H. We define a *graph* as $G\langle V, E\rangle$, where V is the set of all vertices v in the graph, and E is the set of all edges e in the graph. Each vertex v has a type which is non-confusable. Each edge e connects exactly two vertices. We assume these edges are undirected, but there is no reason why they cannot be directed.

All vertices and edges may have labels, which can be extremely useful for understanding the nature of a graph and for increasing the accuracy of search results. However, since we are attempting to find adversarial networks, information may be missing, incomplete, or incorrect. For example, a vertex in *G* may have a label "John", and a vertex in *F* may be labeled "Jonathan". It is quite possible, given our understanding of names, that these two vertices could represent the same person. This shows how labels are *confusable* because they do not need to be exact matches in order to be synonymous. This idea of confusability between labels encourages us to ignore labels at this point in order to focus on the concrete nature of the structure of the graph itself. Though an algorithm could be implemented to deal with these similarities, we do not implement that here. This leaves us with only the notion of the *types* and *edges* of nodes as reliable measures. We will show that this is sufficient information to get good search results for certain parameters of the database *G* and of a query graph *F*.

2. FREQUENCY VECTOR

When we began our exploration of this problem, we determined which parameters of the problem we wanted to attack first. The definitions above eventually were decided upon. The key feature of this description is the presence of types. Imagine for a moment a dataset which models the interactions of a system. Each node may be distinguishable from each other by a few simple traits such as the type of the node and the interactions between this node and its neighbors. A particular student may be represented by a node having 50% of its neighbors as websites and the remaining neighbors as libraries. This particular distribution may be unique to this student, and based on the database, this ratio may be an effective way of classifying each node with an index for fast search later.

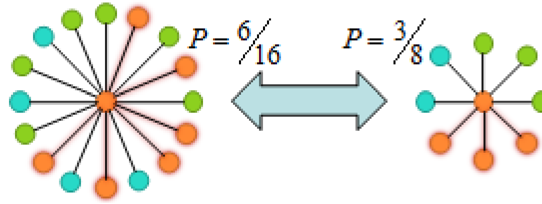

Figure 2.1: Frequency-vector diagram.

Figure 2.1 shows the ratio of neighbor types on the left is the same ratio as that of the neighbors on the right. This information may be statistically relevant based on the database.

2.1 Definition

Each vertex's *frequency-vector* is defined as a vector of length $K = |T|$, where T is the set of all types present in the graph. Let $a_{n,x}$ be the number of neighbors of type n of vertex x. Let $\langle a_{1,x}, a_{2,x}, \ldots, a_{K,x} \rangle$ be the *type-vector* of x. Then the *frequency-vector* of x is the vector

$$\langle \frac{a_{1,x}}{\sum_{i=1}^{K} a_{i,x}}, \frac{a_{2,x}}{\sum_{i=1}^{K} a_{i,x}}, \ldots, \frac{a_{K,x}}{\sum_{i=1}^{K} a_{i,x}} \rangle$$

If a vertex has zero neighbors, we consider it to have a frequency-vector containing all zeros instead.

2.2 Indexing Step

Given a database graph $G\langle V, E\rangle$ we first index the database to make it easier to search later. In order to do this, we calculate a position for each vertex $v \in V$ and store it. We consider that we can represent the *position* of a vertex by the normalized distribution of types of its 1-neighborhood, or *frequency-vector*, as described above. We can then represent the nodes in the graph in a K-dimensional space, and develop its *kd*-tree [5] representation using these vectors as if they were positions. The *kd*-tree representation will yield an efficient search procedure since the size of the database graph is significantly larger than the number of types in use, *K*.

2.2.1 Kd-tree Details

The kd-tree structure is a data structure which is constructed to perform a binary sort across multiple dimensions. At each level of the tree, a dimension from 1 to *K* is chosen and the points are split based on the median of their position in this dimension. Normally the median node is stored at each split, until there are no nodes remaining in the graph. We use a slight variation of the kd-tree structure, where all nodes are stored in the leaves, and the splitting procedure stops when there are 100 (an arbitrary constant) or less nodes per leaf. This modification makes conceptualizing the data structure relatively straight forward, while keeping the implementation relatively simple. The search time of a kd-tree search is roughly O(log n). Kd-trees generally work best when the number of points is far greater than the number of dimensions. A graph with only slightly more nodes than types, for example, will only be slightly better than linear search.

2.3 Selection of Representative Node

The search step begins by a user providing a fragment graph *F* such that the size of *F* is much smaller than *G*. Since *G* is so much larger than *F*, we attempt to first narrow down the locations that we are interested in before attempting a more detailed search. We accomplish this by first calculating the frequency-vectors of the fragment. Next we attempt to select a *representative* node. The particular selection may be done by the analyst before hand, based on his/her intuition, or by a method based on the structure of the fragment. The goal is to identify the node with the most identifiable information, or

perhaps the most relevant information. Once such a node is selected, our search will attempt to locate this node in the database.

2.3.1 Populous Method

The first proposed method of determining a representative node. For each $v \in V$, the representative v is the node with the most neighbors.

2.3.2 Diverse Method

Later, we transitioned to the *diverse* method. For each $v \in V$, let v_N represent the 1-neighborhood. If v_N has more types than any other v, then it is the representative node. If two or more nodes are tied for the most types, the v with more neighbors is the representative node. In our experiments we found that this improved results, and so abandoned the populous-method in favor of this.

2.4 Search Step

Once a representative vertex has been selected, we attempt to locate that node within the database. We search through the kd-tree for the position which identifies the representative-node to the leaf where the point would be located, if it exists. If it does exist and is the same type as the representative-node, the node at that point is the answer of the search. Otherwise, the vertex of the correct type with the smallest distance to the point is the "nearest neighbor", and thus the answer of the search. We use the traditional *kd*-tree traversal method, which requires determining if the splitting points which define the leaf containing the point are closer the point than the nearest node within the leaf. If this is the case, there is a possibility that a node which exists in a different leaf node is actually the nearest neighbor. The algorithm then examines each leaf-node which shares a border closer than the current best distance.

When we compare the two position vectors, we have several options of how to measure the distance. We uniformly weight each dimension, though a non-uniform distribution of weights would also work. Next we sum the weighted distance for each dimension. By using a uniform weighting, the weighted distance between two position vectors reduces to the Manhattan-distance between the two vectors.

$$distance(x, y) = \sum_{i=1}^{K} w_i |x_i - y_i|$$

Once we find the first nearest neighbor, we attempt to find the next closest neighbors until we find a total of 15 nearest-neighbors. Let each of these best 15 results be returned as our best candidates. The idea of returning more than one result is built out of the idea of creating a system which will work with an analyst. Since we are searching for adversarial networks the possibility exists, and could indeed be quite probable, that the first result found in this manner is incorrect. The correct result could perhaps be as close as the second nearest neighbor. Since we do not know how similar two nodes in a graph may end up being, we want to return several potential results which the analyst could then use their expertise on. The constant is arbitrary, however we do not want to return too many results such that the analyst is overwhelmed. We felt that 15 results was reasonable to examine by a human analyst. This number could be tuned to a specific database to return a more optimal number of results based on obtaining the maximum possible correct results within the fewest possible returned results.

Once we have a list of candidates. We move on the more computationally difficult aspect of attempting to find which of these candidates is the best, and sorting them as such. To do so, we take the intermediary step of constructing a subgraph *H* centered on each candidate node by taking the 1-neighborhood around each.

2.5 Scoring Step

The next step of the algorithm is to attempt to locate the best match for the fragment graph *F* from our list of candidate subgraphs. Our goal at this step is to align *F* with each *H* and score the mapping. Then we will sort the subgraphs by this score, and finally return the results. This problem of comparing two graphs and attempting to determine whether they are isomorphic is the same as we described above. Since the problem is difficult (NP-hard), we attempt to estimate the best solution rather than calculate the exact solution, which would take longer.

The exact solution to this problem can be modeled using a recursive backtracking algorithm [6]. Essentially, given a set of nodes, we give them an arbitrary order. We tentatively map the first node of the fragment to a node of the appropriate type in the

subgraph. We then map the second node in the same way, never mapping two nodes in the fragment to the same node in the subgraph. At the end of this procedure, we have a mapping between the two graphs, and can verify whether they are the same using a distance function. At this point, we can unroll the mapping back a step. We now attempt to map the last node with each other possible node, and score each of those. When no possible further mappings exist, we unroll two nodes, and attempt to map the second to last node to a new node. We continue in this manner, scoring and unrolling, until all possible mappings have been tried. The benefit of this procedure is that it is guaranteed to always find the best mapping possible since it tries all possibilities.

This algorithm can be improved. The first improvement we use is pruning of branches. Once the first mapping has been scored, we return this value as the current best mapping. Now, we score partial mappings in addition to completed mappings. When choosing a node to map to, we can add the additional condition that the resulting mapping can be no worse than the current best mapping. This saves time by not searching branches which begin with very poor mappings. The second improvement sacrifices the exact solution in exchange for a speed increase. After we map X nodes using the recursive backtracking scheme, where X is some given value, we use a greedy mapping scheme to map the remaining nodes. This greedy scheme for a node looks at each possible partial mapping for that node in turn, and chooses the partial mapping with the best score. We found that without the greedy step, the backtracking could become much longer than desired, and yet not much recall was lost by using it instead. In this circumstance recall refers to the position of the correct subgraph within the returned list of results.

2.6 Analysis

Once we obtain the sorted list of results, we attempt to grade how well the algorithm did. In a practical setting, for arbitrary inputs, it can be difficult to determine what metrics can be used to determine this. As a result, we created a testing system where the program takes in an argument for how large the fragment should be. It then generates test fragments by selecting a random node in G and growing our fragment from it. Each neighboring node is added with a ¾ probability, and then neighbors of added neighbors

are added in the same way, until the fragment reaches the desired size. Since each fragment graph F is taken directly from the database G, they all retain their ids. We also guarantee that the fragment is a subgraph of G in this way. At this point, we could distort the fragment further, however we found that by choosing nodes with a ¾ probability that the fragment was already distorted enough for our purposes. We now take each candidate subgraph H which was aligned onto F, and take the union of F and H. We then use this equation to get the total overlap.

$$\frac{|F \cap H|}{|F \cup H|}$$

We define this value, which is somewhere between zero and one, to be the ratio of overlap. Note that for the sake of this step we do not consider the alignment between F and H, we only consider the presence of the ids. We consider that each graph must have some minimal overlap with the fragment in order to be counted as a success.

The results generally are not tremendously good in terms of raw success (see Figure 5.4). The algorithm was run on fragments of 20 nodes. Each line represents a different average-degree for the database graph. The x-axis represents the number of nodes in the database. Each test is the average of 1000 trials. Success here is a measure of how often a graph met the minimum overlap requirement within the best 15 results. We can see that for this set of parameters, the success barely reached 60% at best. We will discuss this in more detail in the results section, however this shows that the theory behind frequency-vector is sound.

3. TYPE MATRIX

The second method we present is the *type-matrix* method. During our experiments with *frequency-vector* we became interested in the idea that while the relative frequency of each type might be useful for some applications, an analyst may construct their fragment for a different sort of dataset. For example, the database may keep track of which restaurants a particular node relates to, however the analyst may not have any data on the matter. When the analyst attempts to construct a fragment without restaurants, the lack of restaurants causes each of the frequencies of all other types to increase, and that of restaurants to go to zero. By under-representing just one type, the distance to the correct node is increased in all dimensions. To some extent this can be dealt with through clever use of weights, however this may place a requirement for unnecessary effort onto the analyst. As a result, we developed the unbiased type-vector for this purpose. By not normalizing the type-vector, underrepresentation of one type will only affect the distance to the correct node in that direction. The type-vector method performed well, and was then extended to *type-matrix*, where a node is not only categorized by its type-vector, but also by the type-vectors of each of its neighbors. This method provided its own interesting set of challenges, along with its own benefits and disadvantages.

3.1 Indexing Step

The indexing step for type-matrix method is simpler than that of the frequency-vector method. When the graph is loaded, each node must calculate its position. As before, let $G\langle V, E\rangle$ denote the input semantic graph with each node categorized by a type. Let $a_{n,x}$ be the number of neighbors of type *n* of vertex *x*, where $x \in V$. If there are K discernible types in G, let $\langle a_{1,x}, a_{2,x}, \dots, a_{K,x}\rangle$ be the *type-vector* of *x*.

At this step during the frequency-vector method, we would place each of these positions into a kd-tree. We do not do this for type matrix, because the distance metric which we choose to use is a more involved process. Since each node is represented by its type-matrix, measuring the distance between two nodes requires being able to determine which type-vector from the fragment is comparable to which type vector from the candidate. This process requires at least a cursory mapping between the fragment and candidate subgraph. An attempt could be made to represent this structure in a kd-tree,

though the high dimensionality resulting from every node being characterized by a vector of type-vectors would make use of the structure directly infeasible.

3.2 Search Step

During the next step of the algorithm, an analyst will supply a fragment graph F to the search. The representative-node, *C*, will be selected as it was in frequency-vector. Next we do a linear search through all the nodes. For each node $v \in V$ of type *t*, where *t* is the type of *C*, we compare the type-vector of *v* to the type vector of *C*. Consider for a moment a comparison between two type-vectors. The vectors are identical except that the fragment's vector contains one neighbor of type *A*, while the other contains zero of type *A*. Now consider the same situation except that the fragment's vector contains *n+1* neighbors of type *A* and the other contains *n*, where $n > 0$. In this example, the difference between each is 1, however it seems clear that the presence of the type compared with the absence of the type is more dramatic than the difference between *n* and *n+1*. We consider this to be more important information, and propose that if a fragment's representative-node is connected to a particular type, then it is a significant feature. This begins to get into an assumption that the database has more information than the analyst, however in any situation where data is being entered that the database does not have access to, results will be poorer than when this isn't the case. We feel that having a hard penalty on adding connections to new types will more readily allow us to access to inherent strength of type-matrix. Thus, *v* is a possible candidate if and only if

$$\prod_{i=1}^{K} (a_{i,C} > 0) \rightarrow (a_{i,v} > 0)$$

At this step, we have simply shown that each *v* selected is at least somewhat related to the same things as *C*. The next step is to perform a mapping between the neighbors of *C* and the neighbors of *v*. Since the goal of the search step is not to get a perfect mapping, but simply to attempt to ascertain which candidates should be scored using a more time-intensive method, we do not use any backtracking at this point. Instead we use a simple greedy mapping, as described in the scoring step of the frequency-vector. Once the mapping is established, it is given an initial distance score. As before, the top

15 candidate nodes are passed onto the scoring step. The scoring step works the same as in the frequency-vector method, using a Manhattan distance to score matches.

3.3 Analysis

We immediately see in Figure 5.7 that type-matrix performs much better on this set of tests when compared with frequency-vector. Even the worst result in Figure 5.7 is better than the best result from Figure 5.4. The fact that most of these tests capped out at or near 100% success was very surprising to us. Perhaps even more surprising is that the success rate seems virtually independent of most parameters. Since this algorithm performed so well, we needed to test it on a real world data set. In addition to the random datasets, we constructed a database based on Wikipedia, which will be described more in detail below.

4. Generation

At this point it may be useful to explain the nature of the graphs which we used for testing. For testing we used a set of pseudorandom graphs so we could get an idea of the range of acceptable parameters for which our algorithms would work well. In addition to this, we constructed a graph based on Wikipedia for the purpose of determining whether these results would generalize to real world data.

4.1 Random Graphs

The graph generator takes several arguments to create the pseudorandom graph. First, it takes an integer N which represents the number of nodes. It creates N nodes in the graph, each with zero edges. Then it takes in an integer E, which represents approximately how many edges will connect to any given node. We create a normal distribution centered on the input E, and assign each node a number of edges according to a random point on this distribution. We distribute the edges randomly across the graph, such that no vertex will receive more edges than it is allotted. We chose to make these edges undirected. Next the algorithm takes an integer T, which represents the number of types present in the graph. Each node is assigned a type uniformly at random.

4.2 Wikipedia Database Graph

Random graphs are very useful for testing algorithms and easily altering parameters such as size, or average degree of nodes. However, since we want our system to also work on graphs constructed from real world data, we created one from the ever popular user generated encyclopedia, Wikipedia.org. Each page in Wikipedia consists of a title, a text entry, and optionally semantic information and an infobox containing information which is often contained on similar pages (pages about people may often contain a "place of birth" in this location, for example). In order to construct our graph, we needed the notion of types, so we decided to use only pages with semantic information already predetermined.

We used a website known as dbpedia.org (date last accessed, 11/302010), which stores databases of semantic information pulled from Wikipedia.org, to construct our graph [7]. To accomplish this, we needed to gather several pieces of information for

each page. First, we used the *Ontology Infobox Types* database. We will refer to this as the *type database*. Using this database, we create a node for each page with an ontology and give it a type based on the ontologies presented within it.

The type database is set up in triplets. The first part of each triplet is a webpage containing a unique identifier for the page. The third part of the triplet is a web address containing one ontology tag for that page. Any page appearing in this database has at least two ontologies presented. The first of these ontologies is always *#owl-Thing*, a special tag. We deemed that this level of the ontology structure was not useful since it did not differentiate any page from another. To construct our first Wikipedia-database-graph, we opted to use the *second* ontology presented in the database as a result. We did this because it was apparent by the way the database was set up that the second type given was the most general, but still differentiating, type provided. This gave us a total of 27 types, and approximately 1.47 million nodes in our graph.

Next, to get the edges we used the *Raw Infobox Properties* database. We will refer to this as the *edges database*. The edge database is also setup in triplets. The first part of the triplet is a web address containing the same identifying page name as the first database. This is the page the infobox is part of. The third part of the triplet is a piece of information. It can be a string of characters, a number, or a web address. If it is formatted such that it is the address of a page which we have added to our database during the first step, then we consider the pages described by the first and third parts of the triplet to be linked. For our purposes, we decided to make this graph undirected, since we felt that if a page referred to another page, then the edge may help characterize the receiving page.

Finally, for the sake of speed, we removed all nodes from the database which ended with zero edges. These nodes would be impossible to differentiate using either of the methods presented, and allowing them to remain simply increases memory consumption.

5. Results

We ran several thousand tests for various different parameters and compiled the results together to measure the trends which presented themselves and analyze the results of the algorithms. These tests were run using the Computational Center for Nanotechnology Innovations, using 2.8 GHz Opteron processors with 64 GB of memory. Each test was run for 1000 trials and recorded. Our algorithm takes many parameters, but we did not run a full suite of tests for each combination of each parameter. Unless otherwise specified, the values listed here are the values used in all data shown in this paper. We used 30 types for our random graphs. This number was very close to the number of types in our initial Wikipedia graph (27), so we felt it would be a comparable value. Each of our tests were set to accept a 25% overlap as a success. The most measure of each of our algorithms is how often it counted a success for some set of parameters. As a result, during the scoring step for these tests we used a backtracking depth of two. This would not affect the success, however it would affect how far down the search list you would have to look to find the correct result. This parameter we'll refer to as recall. A recall of *N* would mean that the first success was located at the *Nth* returned candidate graph.

5.1 Random Graph Results

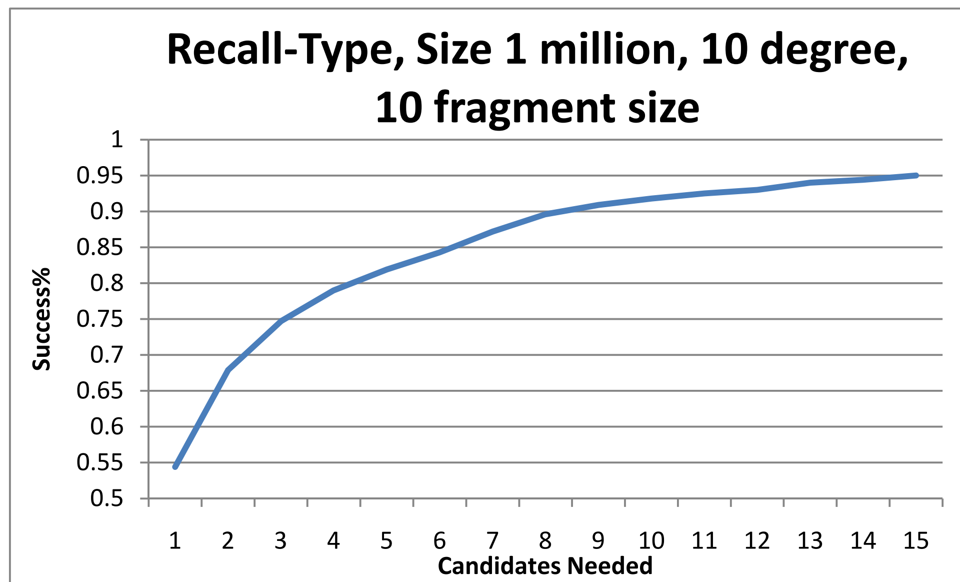

Figure 5.1: Cumulative success as measured as a factor of recall. Size 10 fragments.

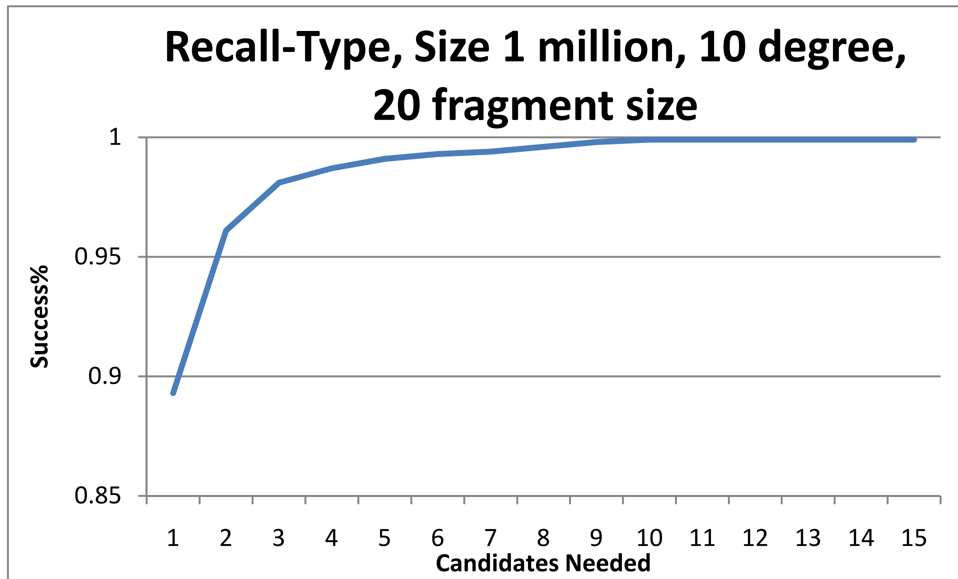

Figure 5.2: Cumulative as measured as a factor of recall. Size 20 fragments.

Here we see why we can get away with using only two levels of backtracking during our scoring step. In the tests diagramed in Figure 5.1 and Figure 5.2 we see that the only parameter which is different is the fragment size. Each of the other parameters are otherwise the same. In Figure 5.1 we see a very smooth distribution of successes, with the most successes in the first result, and each successive result after that attributing less and less to the cumulative success of the results. The success caps out at 95%, but given a few more results, say 20 results, there would be a good chance to come very close to 100% accuracy. However, as we see in Figure 5.2, using a fragment size of 20 makes this curve much more dramatic. By the second graph, there is already a 95% chance of a success. In fact, this curve is not the exception, but closer to the standard for our results on random graphs. Generally we see that after the first two results, very few more successes occur. When using frequency-vector, the cut off is even more stark. As a result, at this point we feel confident that our scoring step achieves its goal of quickly sorting potential matches, since it nearly always moves the correct match to the top of the results.

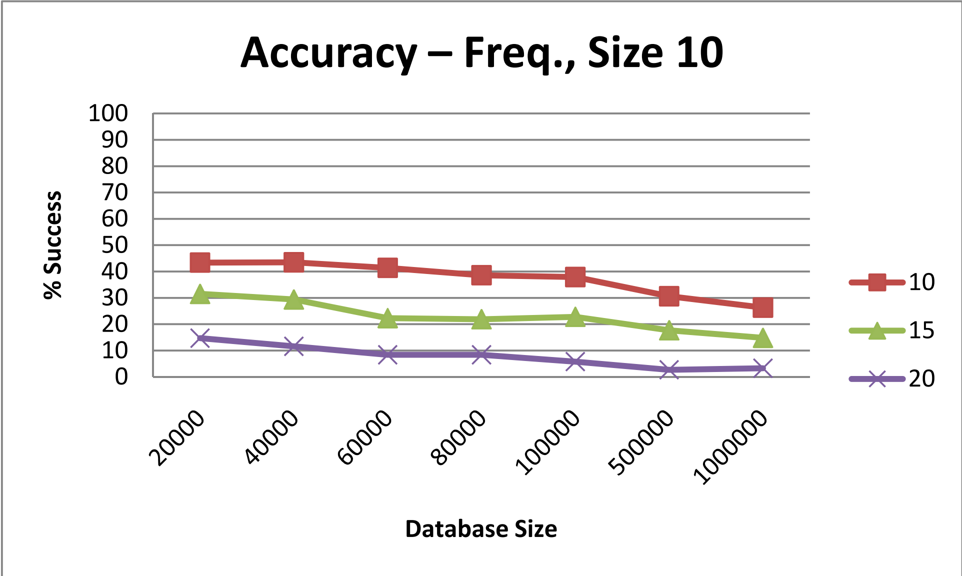

Figure 5.3: Accuracy of frequency-vector across several parameters. Size 10 fragment. Lines are average degree of each node.

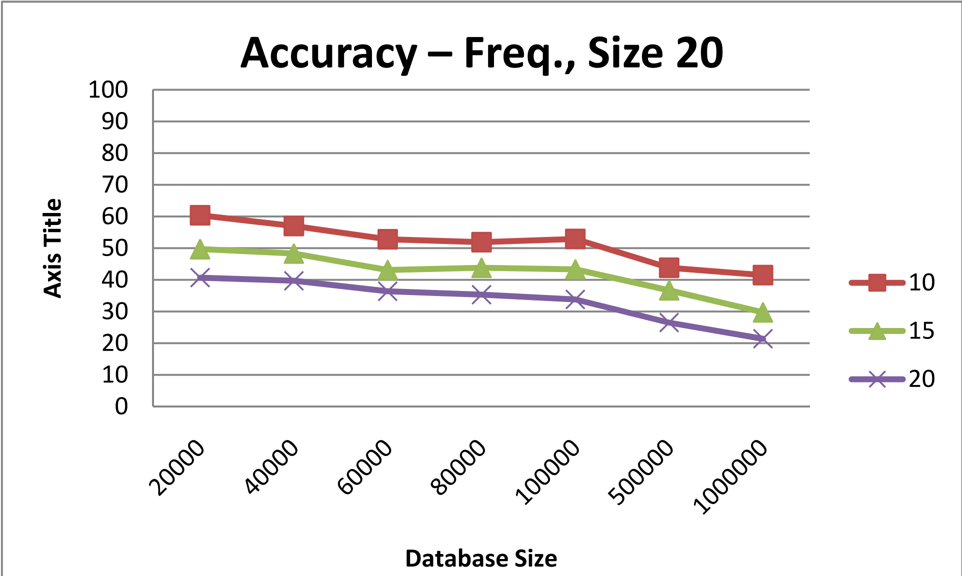

Figure 5.4: Frequency-vector algorithm on randomly constructed graphs.

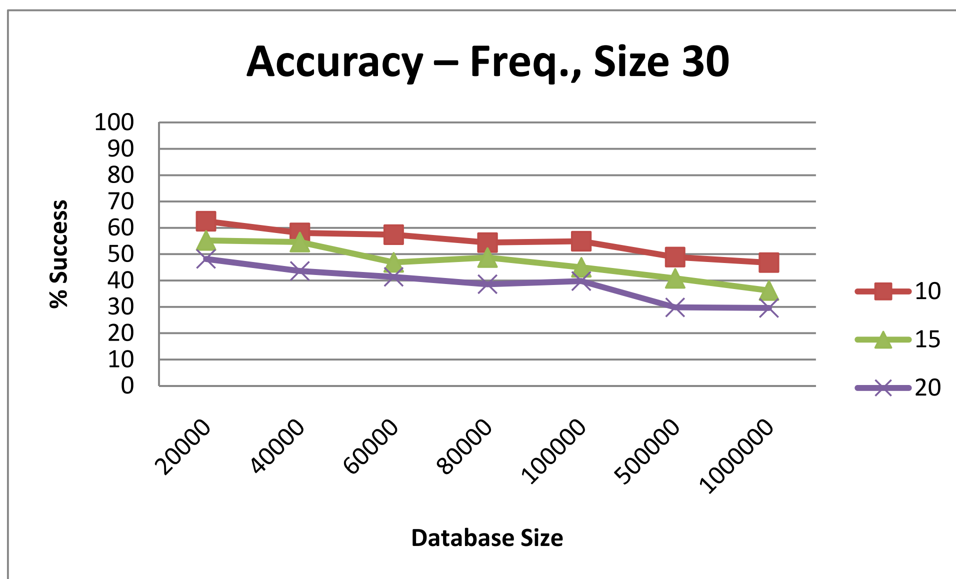

Figure 5.5: Accuracy of frequency-vector across several parameters. Size 30 fragment.

When taking Figure 5.3, Figure 5.4, and Figure 5.5 together we can make some statements about the trends of frequency-vector. In each of these plots we see that frequency-vector works slightly better when the average degree is lower. This can be attributed to the possibility that the normalization involved causes more dramatic repercussions in terms of the identifiability of a node when the number of neighbors is higher. Additionally we see a trend which shows that as the size of the fragment increases the accuracy increases. This gives the fragment more information, so this is not surprising. Each of these plots also shows a trend of a gradual decrease in success as the number of nodes in the database is increased. This is due to the fact that as the number of nodes increases, the number of nodes which have similar descriptors will grow.

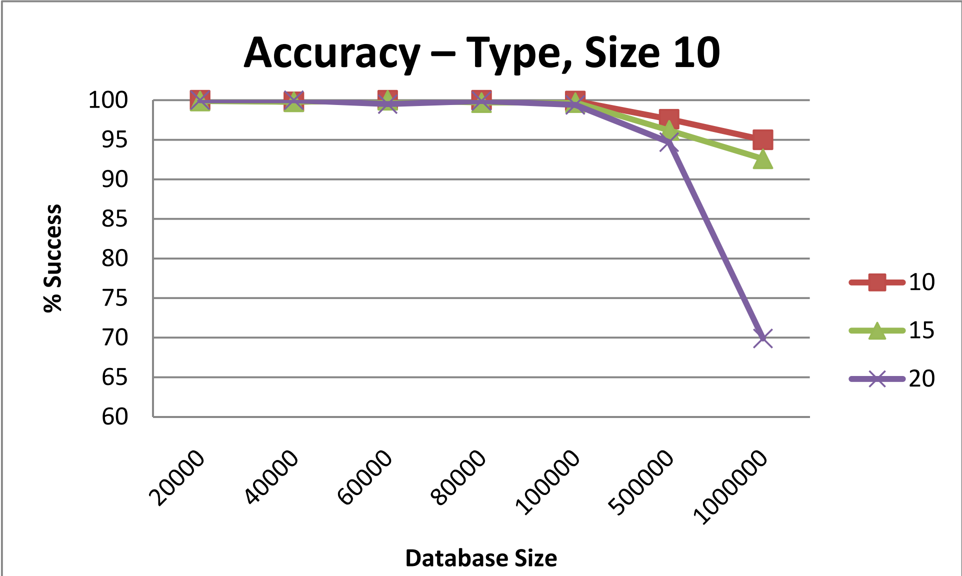

Figure 5.6: Accuracy of type-matrix on size 10 fragments.

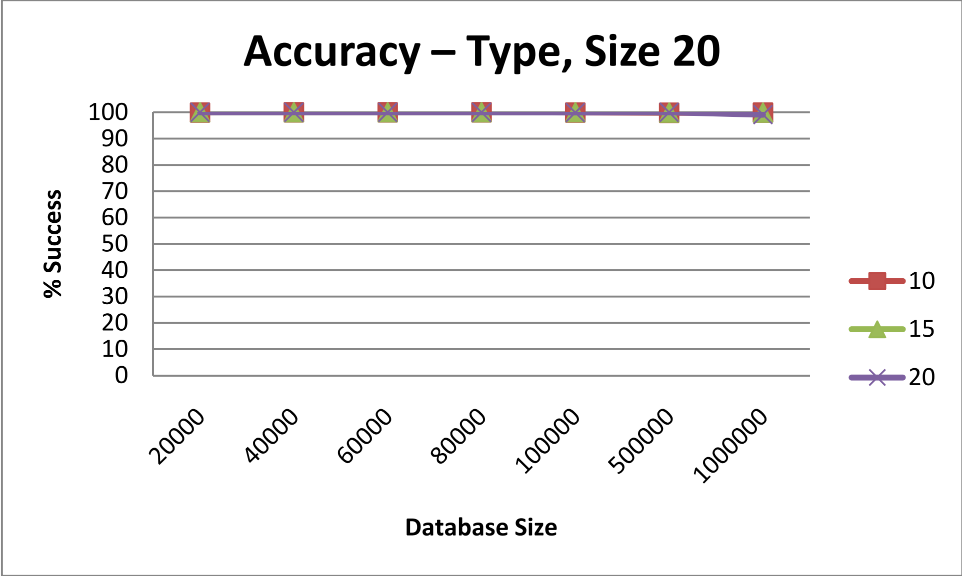

Figure 5.7: Type-Matrix approach. Compare to Figure 5.4.

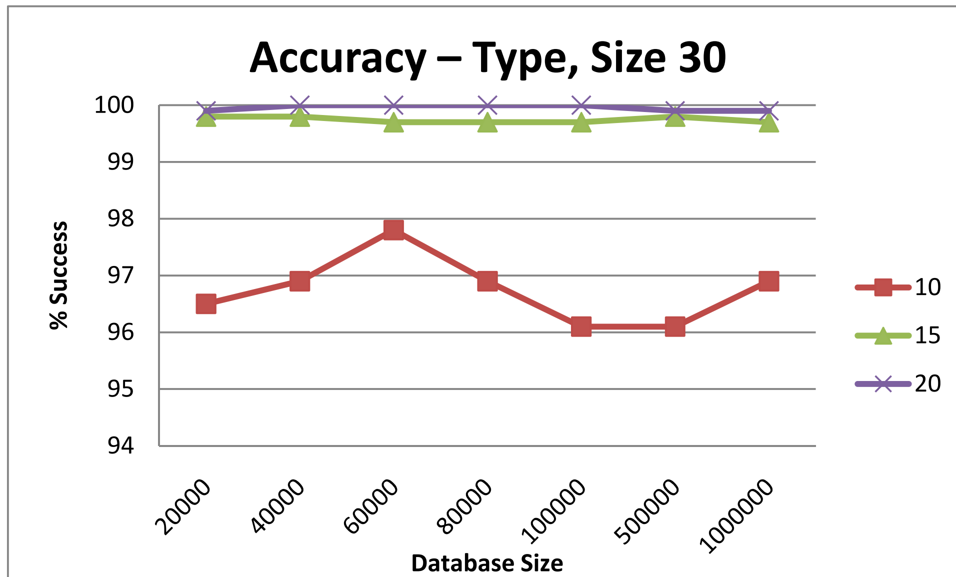

Figure 5.8: Accuracy of type-matrix on size 30 fragments.

Here in Figure 5.6, Figure 5.7, and Figure 5.8 we see the accuracy of type-matrix generally is around 100%. However a few trends of note appear when we magnify the y-axis. We clearly see in Figure 5.8 that for a size 30 graph, having more edges is advantageous in terms of success. This is reasonable, and just the opposite trend as seen in frequency-vector. However, in Figure 5.6 we again see the trend that more edges is worse. This only appears when there are very many nodes in the graph however, which again is strange since neither of the other two type-matrix graph appear to be affected very much by the increase in the database size. This can be explained quite simply however, as resulting from the comparative size of the fragment and the average number of edges. Since Figure 5.6 uses only a size 10 fragment, it is natural that the type vector of such a fragment would not do well to locate its representative node in a database where the average degree is 15 or 20. This only becomes pronounced when a large number of nodes clouds the possibilities. It also should be noted that since the average degree is 15 or 20 (when it is doing particularly bad), that this would imply that when the fragment was constructed most of the neighbors would be likely to be in the 1 neighborhood of the chosen node from the database, and thus would not have very many neighbors themselves. This reduces the effectiveness of the type-matrix search because the algorithm expects that the neighbors of the representative node will also have information to help locate the graph.

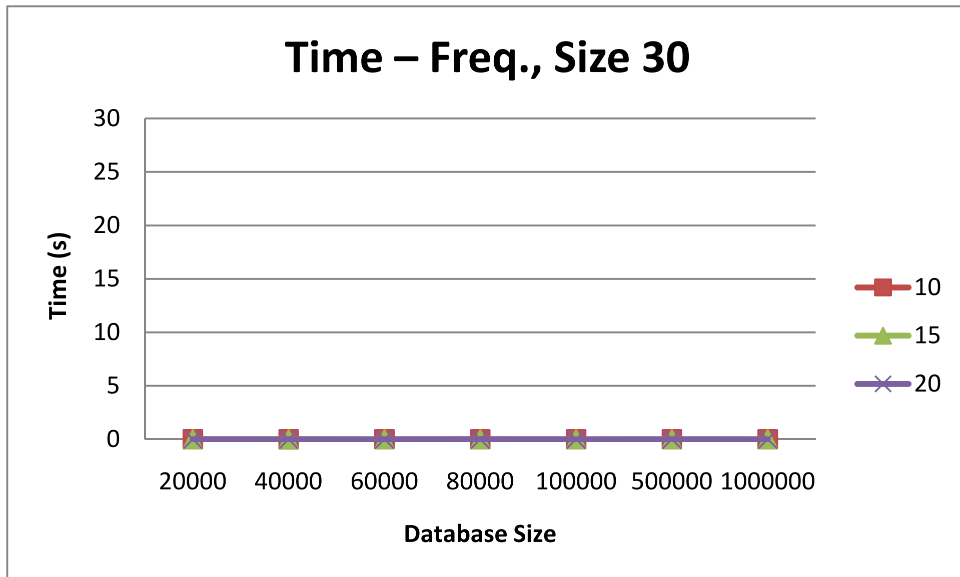

Figure 5.9: Average time for a variety of tests.

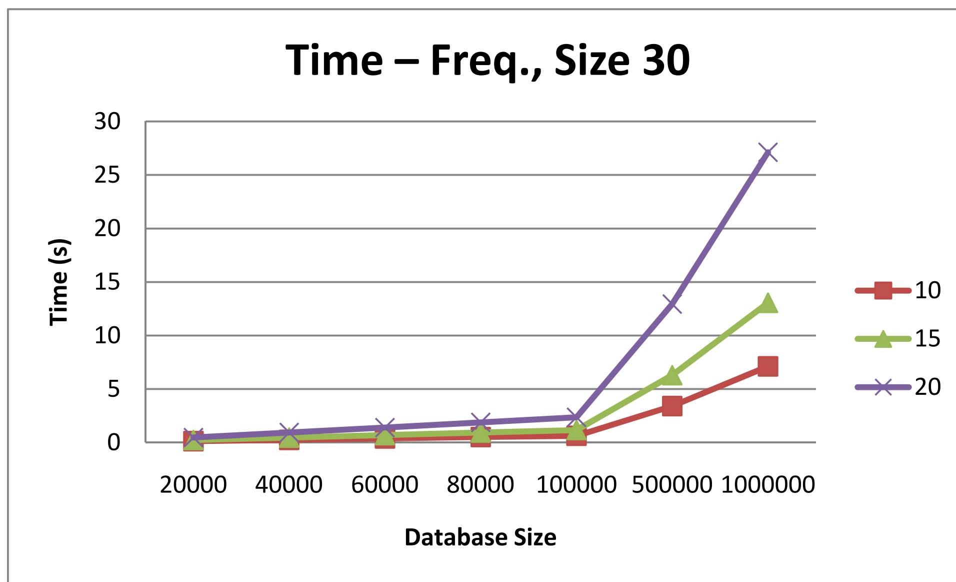

Figure 5.10: Average time for a variety of tests.

In Figure 5.9 we see that frequency-vector searches are quite fast, averaging near zero seconds per search. In Figure 5.10 we see that type-matrix does pay for its improvement in accuracy. The average time for type-matrix is a function of the size of the graph and the number of edges, because it must look through every node of the graph of the appropriate type. For each node examined in this way, it must map against its subgraph, which is a function of the number of neighbors of the node.

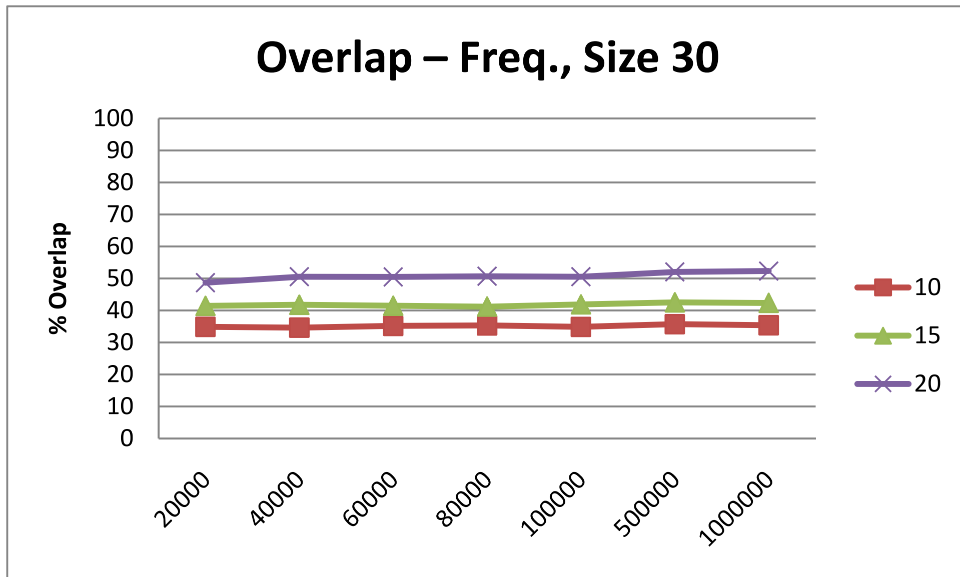

Figure 5.11: Average percent overlap of size 30 fragments.

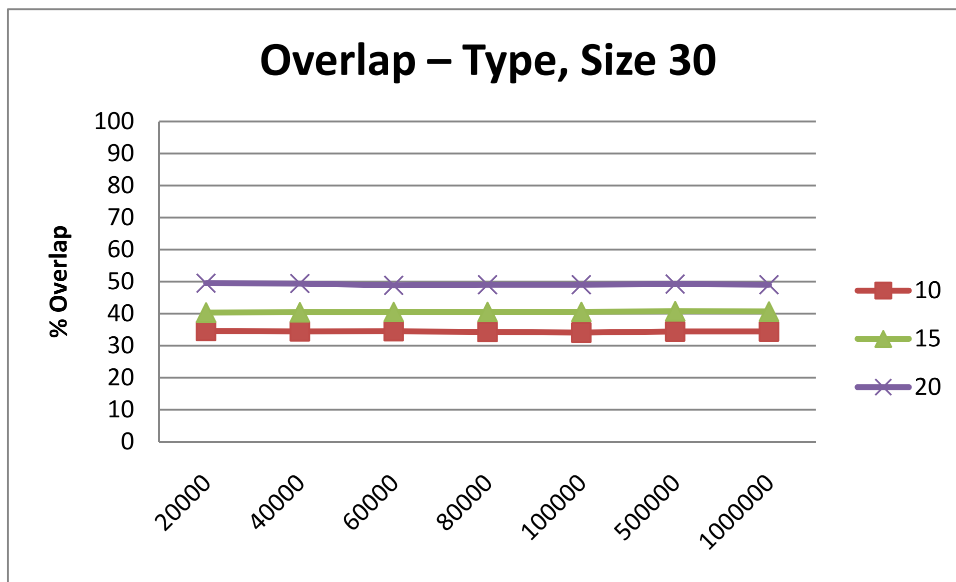

Figure 5.12: Average percent overlap of size 30 fragments.

When we examine the average overlap of Figure 5.11 and Figure 5.12 we notice that they not affected seemingly at all by the size of the database. They should only be affected the by fragment size and by the subgraph size (and thus the average degree) so this is to be expected.

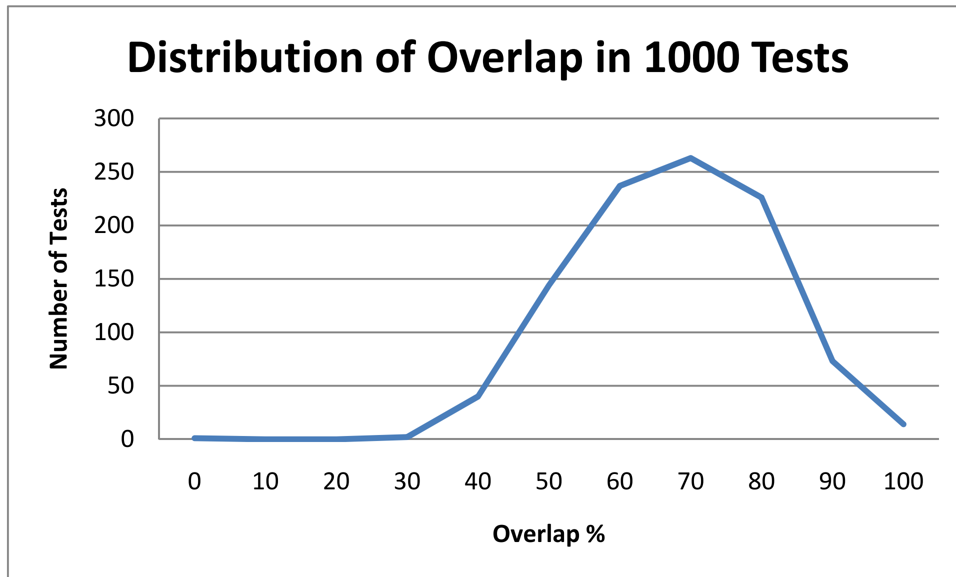

Figure 5.13: Distribution of overlap across 1000 tests using type-matrix.

In Figure 5.13 we see the distribution of overlap across the 1000 tests approximately draws a normal distribution. The database size is 100000, with an average degree of 20 and a fragment size of 20. Tests which had exactly 0% overlap were removed. This distribution shows that the search yields at least a 30% overlap when it overlaps at all. This implies that the graphs are not simply being tangentially overlapped, but generally are either mostly overlapped or not at all. This seems to imply that the search procedure usually correctly locates the region near the representative node.

5.2 Wikipedia Graph Results

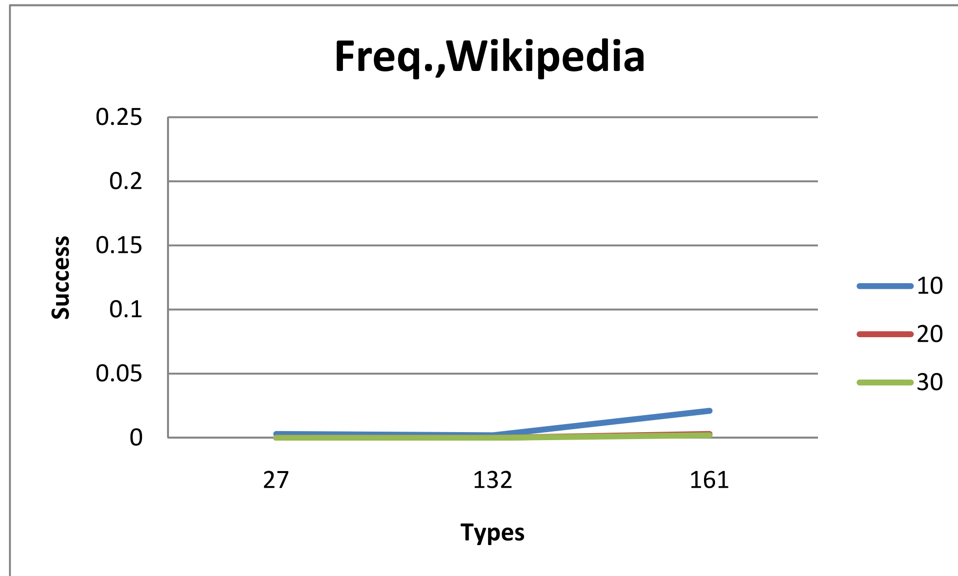

Figure 5.14: Frequency-vector method running on three Wikipedia databases, each with a different number of types. The lines represent 10, 20, and 30 fragment size tests.

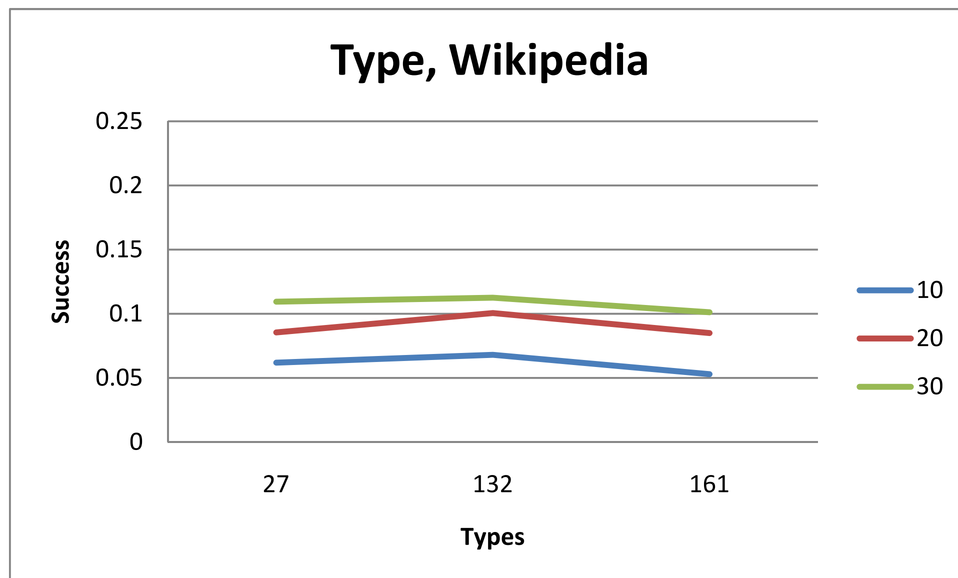

Figure 5.15: Type-matrix method running on three Wikipedia databases. Lines represent fragment size.

We noticed the results we not particularly good despite the presence of 27 types. Compare Figure 5.14 and Figure 5.15 to Figure 5.5 and Figure 5.8, which show results for similarly sized databases and similar number of types. As a result we made a few modifications to some of our algorithms. First, we considered the number of edges in the

graph. The average number of edges per node was ~7, however the distribution of these edges was quite diverse. Some nodes had more than 1000 neighbors, and many had close to 1 neighbor. We came to a conclusion that each of these would cause a problem with the way the fragments were generated. Since they choose neighbors with a ¾ probability, nodes with less than 3 or less edges would average only adding 1 additional edge to the graph each. This could conceivably create a very snake-like graph. Our algorithms assume the existence of one or more nodes which characterize an identifiable piece of information so that a representative node may be chosen. If a representative node only has 2-3 neighbors, it would not be very identifiable, especially given a database with more than a million nodes. This underlines one sort of graph for which these algorithms will not work well. This particular case can be mitigated somewhat by ensuring that fragments attempt to avoid this situation. To do this, we added a feature which changes the fragment generation slightly so that when searching the neighbors of a node with less than four edges, it will take each neighbor with a 100% probability instead of 75%.

Also, the presence of certain nodes with thousands of edges created some large runtime boundaries, especially for type-matrix. Since the algorithm would attempt to make a mapping to every node with the correct type, if these many edged nodes, or pseudo-*stars*, were mapped it would vastly delay the running of the algorithm. As a result we made a few modifications to our algorithms. Firstly we opt to label nodes with more than 1000 edges as a star. Next, when looking for candidates, we do not allow stars to be chosen as potential candidates, as the runtime to map onto them is too great. This is an acceptable loss, since it greatly improves running time. We similarly do not allow the fragment to contain any stars. We do not feel that this is a great problem because if many nodes are connected to the same node, then connection to that node does not make the node much more identifiable, in the same way that saying a person lives in a "country" does not reduce the search space for people very much. We examined which nodes in our database were considered as stars, and indeed found this theory to be correct. Many of the stars were countries of the world. Knowing *which* country a node is from would be useful information, but unfortunately this data is stored as a label and not a type, and thus is ignored.

In addition to the edge distribution, we noticed a further issue when we analyzed the distribution of types in the Wikipedia database. While we did not expect the types of the database to be uniformly distributed, it became clear that the types were disproportionately distributed in the graph. Some type classified as few as a few hundred nodes, while some types classified as many as a few hundred thousand. The expected amount for a uniform distribution would be approximately $\frac{1.47mil}{27} \cong 54444$. In essence, even though we had 27 types, the vast majority of the graph was classified with perhaps 8 types. This was only one of the difficulties this database presented which the random graphs did not. To remedy this issue, in later iterations we attempted to use more of the ontology space which the dbpedia database describes in order to increase the accuracy of our results.

First we counted up each instance of each type within the type database. We also stored a *type-path* for each node. The first type on the path would be *#owl-Thing*, the next would be the type which appears next for this node, and so on. After the final step of the generation algorithm, when nodes with zero edges are removed, we re-type all of the nodes as follows. For each node *x*, if the current type of *x* appeared more than *N* times in the database, then we consider it not specific enough. Change the type of *x* to the next type on *x*'s type-path. This process continues until *x*'s type is specific enough, or *x* has no more types on its type-path. This process is not precisely true to the meaning behind the database, since each ontology provided is not necessarily a more specific descriptor ("scientist" to "physicist"), but could be a sideways jump ("actor" to "governor"). When we used a limit of 70,000 this process increased the number of types in the Wikipedia graph from 27 to 132. This value would help cause the types which had a very large number of members to use types which were more defining. This came along with a slight increase in accuracy. We also used a limit of 30,000 to receive another point of comparison, resulting in a graph with 164 types. Though these approaches did increase the success rate of the Wikipedia database searches, the vast increase of types from 27 to 132 actually seemingly did very little to boost the accuracy.

6. Conclusions

Overall we were very happy with the results of our system. We have shown that both frequency-vector and type-matrix are valid approaches the problem of locating hidden adversarial networks. While it is clear that type-matrix method has a better ratio of success for the graphs presented, this does not necessarily mean that frequency-vector will not have a database for which it will be more accurate. That said, we feel that type-matrix is the more successful approach and will most likely see this extended in future research.

In Figure 5.10, the worst average search time is marked at just over 25 seconds, though the max search time will be longer than that. Under that set of parameters (average degree 20, database of 1 million nodes) the search time may be reaching the bounds of our goals as presented in this paper. If a database were attempted which were larger, or more dense than this, it may be reasonable to say that some optimizations to the algorithms may be needed to keep the time within a reasonable scope. This would help alleviate some of the time constraints which are posed by the Wikipedia graph, where the degree of a node can be extremely large compared with the rest of the graph.

Our method had seemingly low results with the Wikipedia graph, in one part because the data used to construct it is very structured (e.g. every person has a place of birth, so having a connection from person to place is not distinguishing). Our algorithms do not work well when many nodes share the same characteristics of neighbors. For frequency-vector this means the same distribution of types, but in type-matrix this means the same presence of types. That said, if the graph is closer to a uniformly random graph, the algorithm works quite well and fast, as we had hoped to accomplish.

7. FUTURE WORK

Our conclusions show that our software has made a strong step forward towards the problems it sought to investigate. However, the difference in accuracy between our random graphs and our Wikipedia graphs are vast. Further investigation must be done to determine at what point our algorithm begins to see a decline in power as the graph goes from uniformly random to a very structured dataset such as Wikipedia. Such a result may yield more information as to how to construct a graph which will more often produce good search results.

There are several improvements that could be made to the system, including modifying the search step of type matrix by sorting each node's position into a structure, such as a kd-tree, during indexing. This improvement would purely be in terms of time however, and not accuracy.

Leaving aside speed improvements, there are several further paths of research which are available from this point. One prominent improvement is the idea of integrating label semantics into the system. Since currently the system ignore labels due to the adversarial nature of the network which is being sought, the ability to compare labels in such an environment would be a nice addition, and should help deal with graphs which suffer from having very uniform structure.

Currently the system uses the notion of exactly one type per node. The problem presented here could be generalized further to accept more than one type per node, such as that which exists naturally in the dbpedia databases. This may provide new and interesting solutions to a more general form of the problem described in this paper.

LITERATURE CITED

[1] Mark Goldberg. The Graph Isomorphism Problem, in Handbook of Graph Theory, (CRC Press), p. 68-78, 2003.

[2] J. R. Ullmann. An algorithm for subgraph isomorphism. J. ACM, 23:31-42, January 1976.

[3] Chen Chen, Xifeng Yan, Philip S. Yu, Jiawei Han, Dong-Qing Zhang, and Xiaohui Gu. Towards graph containment search and indexing. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 926-937. VLDB Endowment, 2007.

[4] Hilmi Yıldırım, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: Scalable Reachability Index for Large Graphs. PVLDB, 2010: 276~284.

[5] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, Introduction to Algorithms, second edition, The MIT Press, 2001.

[6] Evgeny B. Krissinel and Kim Henrick. Common subgraph isomorphism detection by backtracking search. Softw. Pract. Exper., 34:591-607, May 2004.

[7] Sren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudr-Mauroux, editors, The Semantic Web, volume 4825 of Lecture Notes in Computer Science, pages 722-735. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-76298-0 52.