Abstract

Nowadays, new applications have emerged as a result of the growth and development in technology. This has resulted in an increase in the amount of data that is in the form of graphs. Graphical data has become very important in a number of fields including software engineering, bioinformatics, biology, social networking among others. The need to manage this type of data in large databases has become paramount and therefore, techniques such as subgraph query have become very important for graph data management.

In this project, we are studying an approximate subgraph matching method called SAPPER that aims to find all occurrences of the subgraph query matching in the context of noises such as missing edges. This method utilizes the pre-generated random spanning trees that are used to find the approximate subgraphs and the graph enumeration order that enumerates the subgraphs query that are isomorphic to the query graph. This project includes: an implementation, a discussion about this method and how it works, implementation issues that we faced, and some evaluation results.

1. INTRODUCTION

A subgraph query is one of the most fundamental procedures in managing, processing, and analyzing data in the form of graphs. In some applications such as biological networks, graphs are very large, which makes it difficult to process. In this case, the subgraph matching is used to identify the occurrences of the subgraph query in the database graph. This approach builds on the assumption that the graph database is cleaned and the results will be the exact match to the query graph. SAPPER studies the same problem of subgraph matching in the context of noise, it aims to find all approximate subgraph matches of the query graph in large graph. It builds on the assumption that the set of the approximate matches of the query graph is the superset of the exact matches. A subgraph of the database graph is an approximate match for the query graph if $\text{Dist}(s, q) \leq \Theta$. SAPPER finds all occurrences of the subgraphs query that is no more than or equal to the given threshold $\Theta$. Even though this set of subgraphs could be very large and highly overlapped, the subgraphs enumeration order is used to prune some of these subgraphs before matching them with the database graph. The lexicographical order is used to enumerate every subgraph $s$ in a Depth-First enumeration Order. The subgraph that has at least one match in the database will be given an order, however, the subgraphs that have a prefix that is already enumerated, will not be enumerated again [3]. After the enumeration process, SAPPER matches the queries subgraph with the database graph.

2. RELATED WORK

Querying a graph in a database graph is the process of finding all or a subset of the matches of a given query graph. There are several approaches in subgraph matching which are divided into two categories based on the index usage: The first category focuses on subgraph isomorphic and this kind of method does not use an index for the database graph such as subgraph isomorphism algorithms. The second category is an index-based graph matching that uses an index to maintain the database graph easily. This category can be divided into graph indexing and subgraph indexing. The main goal in the graph index is to find the exact matches of the query graph, while the subgraph indexing aims to find all or a subset of the approximate matches of a given query graph.

Since SAPPER is a subgraph indexing method, our focus will be on the subgraph indexing approach. SAPPER is not the only method in the index-based subgraph matching, there are several related works that are done in this area such as GADDI and TALE. In [1], Tian & Patel propose TALE (a Tool for Approximate subgraph Matching of Large Queries Efficiently) which is an approximate subgraph matching method that
uses a hybrid index structure called Neighborhood Index (NH-Index), to maintain information about every node in the database graph. This method identifies the nodes in the query graph by their importance using some measurement, then starts the matching process by matching the important node first, and then extends the matches to match the whole query graph. While GADDI [2] is another index based approximate subgraph matching method that uses distance based index and enumerates all possible approximate isomorphic graphs to the query graph in large data.

3. DEFINITIONS AND DATASET

This section includes some definitions and the basic dataset example that is used in the rest of the paper.

Definition 1:
\( A(q, \Theta) \) is a set of the subgraph queries that contains all isomorphic subgraphs \( s \) to the query graph \( q \) where \( \text{Dist}(s,q) \leq \Theta \).

Definition 2:
\( G = (V, E) \) is the database graph that consists of \( V \) vertices and \( E \) edges. That stores the data in undirected weighted and labeled graph. Every vertex has unique ID and label.

Definition 3:
\( q = (V_q, E_q) \) is a query graph that consists of \( V_q \) vertices and \( E_q \) edges.

Definition 4:
\( N_1(v, G) \) is a set of vertices that connect directly to vertex \( v \). In different words is the first neighbors to the vertex \( v \).

Definition 5:
\( N_2(v, G) \) is a set of vertices that connect to \( v \) within two edges path or that are \( v \)'s neighbors’ neighbors.

Figure 1: The Database Graph \( G \) and The Query Graph \( q \)

4. SAPPER

SAPPER’s process is divided into two main steps:

- Step 1: The hybrid neighborhood unit (HNU) index is used to store the local information about each vertex in the database graph with the help of the bloom filter to speed up the query processing.
- Step 2: The query processing to identify the approximate matches of the query graph \( q \) in the database graph \( G \), this step includes four other sub-steps:
  - Vertex matching to find all candidates matches for each vertex \( v_q \in q \) in \( G \).
  - Enumerating the query graph’s edges lexicographically from the smallest to the largest based on their occurrences in the database graph \( G \).
  - Generate all possible subgraphs of the query graph that are \( \Theta \) or less edge distance away from \( q \). Then, enumerate all these subgraphs in Depth-First Enumeration Order.
  - For each subgraph, generate all possible random spanning trees possible, then match every sub-graph with the database graph \( G \).

The whole SAPPER procedure in algorithm (8), and in this section, we studied and analyzed each process in detail.

4.1 Hybrid Neighborhood Unit Index

The hybrid neighborhood unit (HNU) is used to store the vertex’s degree, vertex’s label, and neighbors. It is like a table of vertex information with the associated neighbor information. In our implementation of HNU index, we collected the local information for every vertex in the database graph \( G \) including vertex’s degree, vertex’s label, vertex’s first neighbors \( N_1(v, G) \) and the second neighbors \( N_2(v, G) \). The set of the first neighbors \( N_1(v, G) \) includes the direct neighbors of a vertex \( v \), while computing the second neighbors of a vertex is not clear enough in the paper[3], if we have to include the first neighbors or not.

In our implementation, we used a simple method called \texttt{Graphs.neighborListOf(GraphDatabase, v)} in JgraphT library that retrieves the direct neighbors of a vertex \( v \). For the second neighbors we tried two cases: in the first case we only considered the vertices that are within two edges distance from \( v \), and in the second case we included all the vertices that are within one and two edges distance from vertex \( v \). As a result of testing both cases, the second case provided more accurate results than the first case, and we considered the result in the rest of the implementation.

As indicated in the paper, the approximate number of vertices in the set of the second neighbors \( N_2(v,G) \) could be two times the average degree of the database graph \( G \), that is calculated as follows:

\[
\text{Average Degree of } G = 2 \times \frac{|E|}{|V|}
\]

This average increases with the size of the database graph and the set of the second neighbors \( N_2(v,G) \) increases as well. For this reason, SAPPER stores \( N_2(G, v) \) in a bloom filter to ease finding the matching vertices in the database graph \( G \).

Definition 6: Bloom Filter

The bloom filter is a data structure that is used to store and test an element membership in a large set. Basically it contains a binary array and multiple hash functions.

In HNU index we constructed a bloom filter for every vertex \( v \) in \( G \) to store \( N_2(v, G) \) set based on
two parameters as defined in the paper. The number
of bits in the bloom filter equals to \([9.6d^2]\) and the
number of hash functions are seven to reduce the false
positive rate to 0.01. For example, if we want to add
the labels of the vertex V2 in the bloom filter: first,
these labels have to pass through all the hash func-
tions \((h0, h1, ..., h6)\) to get a result that represents
the indices of these labels in the bloom filter, let’s say we
get the following results. \(h0(v2) =3, h1(v2) =4, h2
(v2) =6, h3(v2) =10, h4(v2) =7, h5(v2) =14, h6(v2)
=1\). We go to each of those indices of the bloom filter
B and set them to 1 as shown in figure(2).

4.2 Query Processing

This step includes all processing that SAPPER uses
in the subgraph matching process; which includes ver-
tex matching to find all candidates matches for each
vertex \(v_q \in q\) in \(G\), generate random spanning trees
of \(q\), enumerate all subgraphs in Depth-First enumera-
tion order, and then the final graph matching process.

4.2.1 Vertex Matching

The vertex matching is used to find all matching
vertices for each vertex in the query graph \(q\). A vertex
\(v_G \in G\) is a match for a vertex \(v_q \in q\) if the follow-
ing conditions are satisfied:

- If \(v_q\) label is the same as \(v_G\) label.
- If \(v_q\) degree is less than or equal to \(v_G\) degree.
- If the labels of the vertices in \(N1(v_q, q)\), that rep-
   sents the first neighbors, is a subset of the la-
   bels in the set of \(N1(v_G, G)\) for \(v_G\).
- If the labels of the vertices in \(N2(v_q, q)\), that rep-
   sents the neighbors within two edges distance
   from \(v_q\), is a subset of those of \(N2(v_G, G)\).

In our implementation, we implemented the vertex
matching process as in algorithm (1). The result of the
vertex matching is a set of candidate vertices from the
database graph for each vertex in the query graph.
Figure (3) shows the results of the vertex matching
for our database graph example.

**Vertex Matching Results:**

- **Vertex (1):** [1, 2, 3, 20]
- **Vertex (2):** [6, 7, 9, 11]
- **Vertex (3):** [8, 10, 12, 13]
- **Vertex (4):** [16, 18]
- **Vertex (5):** [15, 17, 19]

Although we found a lot of candidate matches for
each vertex in the query graph, these matches are
based on the local information that is given by the
HNU index, and some of these matches could be false
positives. To reduce the number of matches, we have
to consider the global information by generating ran-
dom spanning trees of \(q\) using the random walk that
is described in details in the next section.

**4.2.2 Random Spanning Tree Generation and
Matching**

Since the spanning tree is a special form of a graph,
in this step of SAPPER, we generated different ran-
dom spanning trees of \(q\) to be able to prune some of
the previous matches. Also, using the spanning trees
to query the database graph \(G\) helps to reduce time
consumption.

**Definition 7: Spanning Tree**

Let \(q\) be an undirected connected graph with \(V\)
vertices and \(E\) edges. A spanning tree of \(q\) is a subgraph
that includes all the vertices of \(q\) and \((V - 1)\) edges.

**Definition 8: Random Walk**

The random walk in an undirected graph \(q = (V, E)\) is a process that starts from a selected random ver-
tex as a prime vertex and then randomly selects one
of its neighbors. This process repeats on the selected
vertex every time until we visit every vertex in \(q\).

**Definition 9: Markov Chain**

Markov Chain is a discrete time stochastic pro-
cess that represents a sequence of random variables
or moves between different states \(S\). In our case \(S = V_1, V_2, ..., V_n\) is the set of vertices in the query graph
\(q\). In this chain the probability of transitioning from
one state to another (from vertex \(v\) to vertex \(w\)) is rep-
resented in a matrix called the probability transition
matrix \(P\). This matrix is defined as follows:

\[
P(v, w) = \frac{1}{d(v)} \text{ if there is an edge between } v \text{ and } w, \quad \text{where } d(v) \text{ is the degree of } v.
\]

\[
P = 0 \quad \text{if there is no edge between vertices } v \text{ and } w.
\]

**How to generate a random spanning tree:**

To generate a random spanning tree, we used a ran-
dom walk on \(q\) that simulates a discrete-time Markov
Chain with a finite sequence of moves between the ver-
tices in \(q\) based on the probability that is defined by a
transition matrix \(P\). \(P(v, v) = 1 / d(v)\) is the proba-
bility of moving from one vertex \(v\) to another vertex
(w). Every time we reached a new state or vertex,
the probability will be based on the current state, not
on the previous one. This is how the Markov chain
works with the spanning tree. In our case, the set
of vertices in the graph, \(S = V_1, V_3, V_5\) is the set of
states and with undirected graph, the probability of
moving from v to w is the same as the probability of moving to vertex v, which means the Markov chain is symmetric.

Due to the lack of details in the paper, we could not reach to the exact technique that they use to generate the random spanning trees to get $|V(q)| + 1$ unique trees. Furthermore, the paper did not give any details about the method that they used to calculate the number of spanning trees, and if they consider the reversible tree as a different tree or not. We found that the number of the spanning trees of a graph could be different based on the structure of the query graph. A complete graph $K_n$ is a graph in which each pair of vertices is connected to another while the cycle graph $C_n$ is a graph that contains a single cycle through all the vertices. The number of spanning trees in the cycle graph $C_n = n$, where $n$ is the number of vertices while in the complete graph is $K_n = n(n-2)$. In our implementation, we generated the random spanning trees using the algorithm (2) and considered the number of the spanning tree of a graph equal to $|V(q)| + 1$ as a fixed number. We also assumed that they consider the distinct spanning trees of the query graph $q$ and the non-reversible tree.

The figure below shows some of the generated random spanning trees for a subgraph of the query graph using algorithm (2).

![Random Spanning Trees](image)

**Figure 4: Random Spanning Trees of The Subgraph**

After we constructed all the spanning trees of a subgraph query, we matched these trees with the graph database $G$ based on the tree matching algorithm (3) that is described in the following section.

**Tree Matching Process:**

Since the tree is another form of a graph, matching a tree with the graph database is more efficient and simpler than matching a complete graph. This approach is based on the subgraph matching property:

**Definition 10: Subgraph Matching Property**

Given a query graph $q$ and a database graph $G$, if $q'$ is a subgraph of $q$, then for any matching graph $q''$ of $q$ in the database $G$, $q''$ must contain a match of $q'$.

The tree matching process has a high pruning power to prune some of the vertex matches that we got from the vertex matching process. This process starts by selecting a random vertex from the query graph $q$ to be a prime vertex in all the spanning trees. Then, based on the prime vertex it matches, every vertex in the tree with the graph database based on the candidates set for each vertex. The paper[3] does not give details about the tree matching algorithm and in our implementation, we used algorithm (3). In this algorithm for every vertex we tried to match each vertex matches’ neighbors with the next vertex’s matches to locate the vertices that are near to each other in the graph database. This process is performed in Depth-First manner until we matched the whole tree.

**Definition 11: Depth First Search (DFS)**

Depth-first search (DFS) is an algorithm used to traverse a graph starting from the prime vertex that is considered as a root and goes as deep as possible until it reaches the leaf to visit every vertex in the tree. As described in the pseudo-code below:

```plaintext
DFS(Prime Vertex v)
{ visit-Vertex(v);
  for each neighbor u of vertex v
    if u is not visited
      { DFS(u);
        add edge vu to the tree Ti }
}
```

After applying this algorithm to our database example, we are supposed to have an accurate set of matches for each vertex in the tree represented by $M(v, Ti)$, but we did not get a satisfied matching result, because of the lack of information in the paper.

Until this process, we faced two confusing parts: The first part is how to find the set of $N_2(v, G)$. As we discussed previously, we have two possible ways to compute the neighbors in $N_2$: by considering only the vertices that exist within two edges distance from $v$ or considering all the vertices that exist around vertex $v$ that include first and second neighbors. The second part is in the vertex matching process. There are four conditions in the vertex matching process. Condition 3 and 4 are not stated clearly enough whether they have to be satisfied at the same time or considered one of them enough for two vertices to be a match. Also, these two confusing conditions affected the results in the tree matching phase.

From these two unclear points, we derived four cases:

A Consider a vertex that belongs to the set $N_2(v, G)$, if it is within two edges distance only.

B Consider a vertex that belongs to the set $N_2(v, G)$, if it is within two edges distance or less, which means $N_2(v, G)$ includes $N_1(v, G)$ as well.

C Consider a vertex $v_G$ is a match to a vertex $V_q$, if both satisfies conditions 3 or 4.

D Consider a vertex $v_G$ is a match to a vertex $V_q$, if they satisfy conditions 3 and 4 at the same time.

We tried all possible combinations of these four cases in our implementation and compared the results to select the one that is more reasonable:

1. A and C
2. A and D
3. B and C
4. B and D
After we had applied all these cases, we found that case 3 gave more accurate matches in both vertex matching and the tree matching results. For this reason, we considered the results that we got from this case in the rest of our implementation.

### 4.2.3 Query Graph Enumeration Order

In this process, we first enumerated the edges in the query graph lexicographically based on their occurrences in the database graph starting from the smallest to the largest. Then, we generated all possible subgraphs of the query graphs and enumerated them in Depth-First enumeration order for the final matching process.

**Query Graph’s Edges Enumeration Process:**

In our implementation, we assigned a lexicographical order for each edge in the query graph $q$ based on the number of edge’s matches in the database graph $G$. The order of the edge assigned lexicographically from the smallest to the largest according to the number matches. For example, if there are two edges $e_1$ and $e_2$, and $e_1$ has a smaller number of matches than $e_2$ ($e_1 < e_2$), then we give $e_1$ a smaller order than $e_2$. However, if there are two edges that have the same number of matches, the order assigns arbitrarily to those edges. We used the DFS to traverse the query graph $q$ to count the number of matches for each edge. For example, we started with vertex $v_1$ and retrieved $v_1$’s matches set: let’s say $M(v_1) = x_1, x_2, x_3$. For each vertex connected to $v_1$ in the query graph $q$ for example; $v_2$ and $v_2$’s matches set is $M(v_2) = y_1, y_2$. We matched all $x_1, x_2$ and $x_3$’s neighbors with $v_2$’s matches. In this case we are trying to see if the edge between $v_1$ and $v_2$ actually exists in the graph database $G$. The number of matches between these vertices is the number of occurrences of that edge in the graph database. This process is explained clearly in Algorithm (4).

**Result:** The lexicographical order of the query graph’s edges.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Number of occurrences in $G$</th>
<th>Lexicographical Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1</td>
<td>e1</td>
</tr>
<tr>
<td>45</td>
<td>2</td>
<td>e2</td>
</tr>
<tr>
<td>35</td>
<td>3</td>
<td>e3</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>e4</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>e5</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>e6</td>
</tr>
</tbody>
</table>

Figure 5: The Query Graph’s Edges Enumeration Order.

In our implementation, we gave every edge an id that represents its order. These ids start from 1 to $|E(q)|$, which is number of edges in the query graph $q$ as shown in table in figure (5).

After we determined the order of the edges in the query graph $q$, we generated all subgraphs of the query graph that contain a different set of edges and enumerated them lexicographically smallest to largest using the Depth-First enumeration to start matching these subgraphs with the graph database. To get all possible subgraphs “sequences”, we considered the set of edges in the query graph as a sequence of edge $(e_1 e_2 e_3 e_4 e_5 e_6)$, then generated all possible subsequences of that sequence. These subgraphs called $AI(q, \Theta)$.

**Subgraphs Enumeration Order:**

Since SAPPER’s goal is to find all approximate matches of the query graph $q$, it matches all the subgraphs in the set $AI(q, \Theta)$, where each subgraph matches with one or more subgraphs in the database graph $G$. The number of subgraphs in this set are very large and highly overlapped. The approximate number of subgraphs could be $O(m^n)$, where $m$ is the number of edges in $q$. In our case, if $\Theta=2$ then $m = 36$ subgraphs; that includes the subgraphs within one missing edges ($m^0 = 6$) and non missing edges. SAPPER aims to prune some of these subgraphs as early as possible before matching the database graph $G$ by enumerate these subgraphs in Depth-First enumeration order.

**Depth-First Enumeration:**

SAPPER uses Depth-First enumeration process to enumerate all subgraphs $AI$. This process starts from the smallest lexicographical sequence of edges that could be a subgraph of the query graph within $\Theta$ edge distance from $q$.

In our example, if $\Theta=2$, we started with $q_1= e_1 e_2 \ldots e_l$ ($l = |E(q)| - \Theta$), and if we found at least one match of this sequence (subgraph) in the graph database, we enumerated this subgraph and continued adding an edge with the smallest lexicographical order after $e_l$ to form a new sequence that represents a new subgraph let’s say $q_2= (e_1 e_2 \ldots e_l e_{l+1})$. If we found that $q_1$ is a prefix for $q_2$, then the set of the matches of $q_1$ is the superset of $q_2$’s matches. This will reduce the scope of the search in the graph database to find $q_2$’s matches. Adding an edge to the current subgraph is called LEXI Next, that continues until no more edge can be added to the subgraph (sequence) or no more matches can be found in the database. Since the graphs that contain $s$ as prefix are overlapped with the graph $s$, we only enumerated graph $s$ if we found at least one match in the graph database. In case we did not find any matches for the current sequence (subgraph), we skip all the sub-graphs that contain this sequence as a prefix, and get started with a new sequence that does not contain $s$ in the prefix for example, $e_1 e_3, \ldots e_l$. This procedure is called LEXI Jump, it continues until we reach the last sequence that could be a subgraph $e(l-2) e(l-1) e_l$.

Using the Depth-First enumeration order we could reach to every subgraph in $AI(q,\Theta)$ at least once.

In the paper[3], the authors use LEXI Next and LEXI Jump to enumerate the subgraphs in Depth-First enumeration and after we analyzed these processes, we found that they are basically used for sorting the subgraphs (sequences) based on their edges’ lexicographical order as shown in the figure (6). In our implementation, we considered the set of edges as a sequence of characters in a string to get all possible combinations of these edges taking in to consideration the lexicographical order for these edges. we created
all possible subgraphs first then we matched them in order.

In our query graph example, the enumeration process started with the smallest sequence “subgraph” that contains a set of edges with smallest lexicographical order \((q1 = e_1 e_2 e_3 e_4)\) where \(\Theta = 2\), and ended with largest subgraph that includes edges with largest lexicographical order \((q = e_3 e_4 e_5 e_6)\). Figure (7), shows all candidates subgraphs of the query graph with less than or equal \(\Theta\) missing edges with their enumeration order. In the following section we explained how SAPPER uses this order.

4.2.4 Graph Matching

In this phase, we processed all the subgraphs queries top-down and left-to-right order. For every subgraph we have to check several cases before matching the subgraph with the database graph:

First: subgraph Connectivity: We have to check if the subgraph is connected or not using the DFS to traverse the subgraph and check if all the vertices connected to each other. If the number of vertices that has been visited by the DFS is equal to the number of the vertices in the tested subgraph, then the subgraph is connected otherwise it’s not. In this case, we only considered the connected subgraphs.

Second: Spanning Tree: we have to check if the subgraph can have a spanning tree or not. In this cases there are two scenarios to consider before searching for matches:

In the first scenario, we have not searched any prefix of the subgraph query \(q_1\). In this case, we have to check if the subgraph can have a spanning tree or not by comparing the number of edges to the number of vertices in the subgraph. If the number of edges is greater than or equal to the number of vertices in the subgraph, then the subgraph can have more than one spanning tree. In this case, we generated some random spanning trees for that subgraph using Random Spanning Tree Algorithm (2). Then, searched for matches for each spanning tree using the Tree Matching Algorithm (3). For example, if the subgraph has \(T_1, T_2, \ldots, T_r\) spanning trees and \(M(v, T_i)\) for all \(1 \leq i \leq r\) are the matches of the prime vertex \(v\) in each
spanning tree. As a result, the intersection of the prime vertex’s matches in all the spanning trees is used as a starting point to find the exact match of the subgraph query \( q \). In the second case, if the number of edges is \( |V(q)| + 1 \), then the subgraph is itself a tree, in this case, we searched for its matches using the Tree Matching Algorithm (3).

The second scenario, we already have some matches of \( q_1 \)’s prefix. In this case, we extended the matches set of the prefix to search for the edges that belong to the current subgraph and not to the prefix. For example, if the first subgraph is \( q_1 = e_1 e_2 e_3 e_4 \) and the next is \( q_2 = e_1 e_2 e_3 e_4 e_5 \), as we can see \( q_1 \) is a prefix for \( q_2 \), and if we found some matches of \( q_1 \) in the database graph, then we extended the matches of \( q_1 \) and search only for the edges that belong to \( q_2 \) and not exist in \( q_1 \), that is \( e_6 \).

**Example: How this process works.**

SAPPER processes these subgraphs top-to-bottom then left-to-right. As shown in figure (8), the first subgraph is \( q_1 = e_1 e_2 e_3 e_4 \).

This subgraph is a connected subgraph and can have a tree, in this case, we used the tree matching algorithm (Algorithm 3) to find the matches for this tree in the database graph.

The results from the tree matching algorithm are:
- Vertex (1) : \([1, 2, 3, 20]\)
- Vertex (2) : \([6, 7, 9, 11]\)
- Vertex (3) : \([8, 10, 12, 13]\)
- Vertex (4) : \([16, 18]\)
- Vertex (5) : \([15, 17, 19]\)

These results represent the set of candidate vertices for each vertex in the query subgraph \( q_1 \).

Then, for each match of \( q_1 \) in \( G \), we mapped the matches vertices to the vertices in \( q_1 \). In this process, we mapped the query graph edges using the previous results to find the candidate edges that match the edges in the query graph. The results from the mapping process are shown in figure (9).

![Figure 9: e1 e2 e3 e4 Matches in G](image)

The results after mapping process by edges as follows:
- Match 1: \([9, 16], [16, 17], [10, 17], [3, 10]\)
- Match 2: \([11, 18], [18, 19], [13, 19], [20, 13]\)

After identifies the matches of the subgraph \( q_1 \), we continued adding one edge, which is \( e_6 \) to \( q_1 \) to get a new subgraph that is \( q_2 = e_1 e_2 e_3 e_4 e_5 \). Again, we checked this subgraph connectivity and the ability of having spanning trees. In this case, as we can see that the subgraph \( q_2 \) is a connected subgraph and has a prefix of another discovered subgraph that is \( q_1 \). In this case, we used the previous matches of \( q_1 \) to search for the undiscovered edges that exist in \( q_2 \) only using algorithm(6).

The results from this process are shown in figure (10).

![Figure 10: e1 e2 e3 e4 e5 Matches Based of The Prefix’s Matches](image)

After we determined the matches of \( q_2 \), we added one edge \( e_6 \) to the current subgraph \( q_2 \) to get a new subgraph that is \( q_3 = e_1 e_2 e_3 e_4 e_5 e_6 \).

This subgraph will follow the same process as the previous one, we only need to search for \( e_6 \) in the previous matches of \( q_2 \). The results are shown in figure(11).

![Figure 11: e1 e2 e3 e4 e5 e6 Matches Based of The Prefix’s Matches](image)

When we found the matches of \( q_3 \), we cannot add any more edges to the current subgraph \( q_3 \). In this case, we jumped to the next subgraph that is not a super-graph of \( q_3 \). For example \( q_4 = e_1 e_2 e_3 e_4 e_6 \). This subgraph can have more than one spanning trees, in this case, we have to find all the spanning trees first, then match the trees with the graph database \( G \).

We have to check every subgraph in the Depth-First enumeration order because in some case the non-connected subgraph can have a connected supergraph, an example for that is shown in figure (12).
Map The Matches With The Database Graph:
In this process, we considered the two previous scenarios. If the subgraph has no prefix, then the mapping process consists of two steps: first, mapping the matches between the subgraph query and the database graph to identify the matches for each edge. This process helps to determine the candidate edges for each edge in the subgraph query. In the second step, we mapped these matches to find a complete subgraph match in the database graph.

In our implementation, we implemented a method called Mapping-To-Find-Exact-Edge’s-Match. In this method, we mapped the edges in the subgraph query to the edges in the database graph to identify the candidate edges. The whole process is shown in Algorithm 5, for each edge in the subgraph query, we compared the edge source direct neighbors’ matches with the edge target’s matches. Every time we found a match between these two sets of vertex, that means there is an edge between the edge source and target. Then, if this edge exists in G, we compared its label with the label of the edge in the subgraph query. If these two conditions are satisfied, then the edge in G is a match for the edge in the query graph. We continued matching the edges until we found matches for every edges in the subgraph query.

For the second step, we implemented a method called Perform-The-Final-Matches this method created a Cartesian product out the edge’s matches from the previous step to get all possible paths. This method is built based on Algorithm 7.

For the second scenario, if the subgraph has a prefix, we implemented a method called Search-For-Undiscovered-Edges, this method uses Algorithm 6 to search for undiscovered edges in the prefix’s matches set.

In this method, we mapped the vertex’s matches for both edge source and target for the undiscovered edge with the prefix’s matches results. In this process we are trying to find matches for both edge source and target taking into consideration the edge label. If
we find both edge source and target in the previous matches that means there is a match for that edge. We continued searching for matches for undiscovered edges in all previous matches.

5. DATASETS
We used two datasets with different graphs structure to analyze the performance of SAPPER:
1- Our generated dataset.
2- A biological dataset for aids.

5.1 Dataset 1:
This dataset consists of 65 vertices and 129 edges with three district labels, and the query graph has 6 edges and 5 vertices with three distinct labels.

5.2 Dataset 2:
This dataset is a biological dataset for aids that consists of 500 vertices and 500 edges in the database in 20 graphs, and 26,360 vertices and 24,000 edges in the query graphs in 4000 graphs. We used this dataset to identify some cases that affect the performance of SAPPER. In our experiments, we queried 20 database graphs and 4000 query graphs, the total number of the queries is 80,000 query process, and compared the results of the exact matches that SAPPER can retrieve with the exact matches of GraphQL. Also we used this dataset to analyze the relation between the query time and several parameters.

6. EXPERIMENTAL RESULTS
In this section, we analyzed the performance of SAPPER using query graphs with different number of vertices and different thresholds.

In dataset 1, we queried this dataset with $\theta = 0$, and we did not get any exact match of the query graph in the first run. However by querying the database several times we could retrieve some matches. With this dataset the query graph is a complete graph with five vertices and six edges. In this case SAPPER generates random spanning trees that are subgraphs of the query graph then uses the tree matching algorithm to match every tree with the database graph $G$, then combines the results to get an exact match of the query graph. In this process, the probability of having the exact match of the query graph is related to the probability of including all the edges in the random spanning trees, and that’s why after several runs we may find some matches.

However, when $\theta > 0$, SAPPER was able to retrieve all the approximate matches of the query graph that has $\theta$ missing edges or less. Since we have six edges in the query graph, the maximum number of missing edge is 2. In our experiment we queried the database graph with $\theta = 0$, that retrieves the exact matches of the graph, $\theta = 1$ that retrieves the approximate matches with one missing edge, and $\theta = 2$ to get the approximate matches that have two missing edges or less. The number of matches with each one of these thresholds are shown in the table in figure (15) and figure (14) shows the approximate matches, every set of colored vertices is an approximate match of the query graph.

### Table: Number of Exact Matches

<table>
<thead>
<tr>
<th>Vertices in q</th>
<th>GraphQL</th>
<th>SAPPER</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>58251</td>
<td>55758</td>
</tr>
<tr>
<td>8</td>
<td>9909</td>
<td>8783</td>
</tr>
<tr>
<td>12</td>
<td>1553</td>
<td>1386</td>
</tr>
<tr>
<td>16</td>
<td>97</td>
<td>64</td>
</tr>
</tbody>
</table>

Figure 13: Number of Approximate Matches with Different Thresholds.

Figure 14: The Approximate Matches of $q$ with $\theta = 2$

6.1 SAPPER and The Exact Matches:
We tested the efficiency of SAPPER to retrieve the exact matches of the query graph comparing to GraphQL. In our experiment, we queried SAPPER with $\theta = 0$ and We found that SAPPER may not be able to identify all the exact matches of the query graph directly in some query graphs. After we had compared the exact matches in the query graph one by one, we could identify that with a query graph that has more than one spanning trees, SAPPER was not able to retrieve all the exact matches for the same reason that we clarified previously. However, in case if the query graph cannot have more than one spanning tree, SAPPER was able to retrieve all exact matches of the query graph. SAPPER took four hours to find 24640933 exact matches in 10,000,000 queries, while graph GraphQL took less than five minutes to retrieve 25111606 exact matches. The table in figure (15) represents the number of exact matches for part of the database 2 that include 20 database graphs and 1000 query graphs in each category in both SAPPER and GraphQL.
6.2 The Approximate Matches and The Threshold:

We studied the relation between the number of approximate matches and the threshold, and we found that the number of approximate matches with $\theta \geq 1$ increases by the number of missing edges as shown in the table below. Also, we can see in figures (16) that with the query graph with four vertices, the number of matches is the same even with different thresholds, that means there are no approximate matches for these set of query graphs.

| Threshold | $|V(q)|=4$ | $|V(q)|=8$ | $|V(q)|=12$ |
|-----------|----------|----------|----------|
| $\theta = 0$ | 55758 | 8783 | 1389 |
| $\theta = 1$ | 55758 | 18181 | 6221 |
| $\theta = 2$ | 55758 | 18184 | 6318 |
| $\theta = 3$ | 55758 | 18179 | 6318 |

![Figure 16: The Approximate Matches and The Threshold](image)

6.3 The Query Time:

We analyzed the query time in SAPPER on different parameters, and found that the query time is affected by the number of vertices in the query graph $q$, the number of distinct labels in the database graph $G$, and the threshold.

6.3.1 Number of Distinct Labels in The Database:

We studied the relation between the query time and the number of distinct labels in the database graph by filtering the database graphs based on the number of distinct labels on the vertices into three sets: graphs with one label, two labels, and three or more labels. Then we queried these databases several times to get the average of the query time for query graphs with 4, 8, and 12 vertices. The result of our experiment was as shown in figure (17). We can see that increasing the number of distinct labels in $G$ decreases the query time, but increasing the number of vertices in the query graph increases the query time even with the some number of distinct labels. The reason here, is that if the database graph has distinct labels, the number of candidate matches in the vertex matching phase will decrease and that will reduce the time taking to prune the vertices in the tree matching process. This will reduce the number of matching edges as well when it comes to the mapping process.

![Figure 17: Distinct Labels in G and The Query Time](image)

6.3.2 Number of Vertices in The Query Graph:

In SAPPER increasing the number of missing edges will increase the number of the subgraphs that need to be enumerated and match with the database graph. The table in figure (18), shows the relation between the number of the vertices in the query graph and the query time. Each row shows the increasing of the query time with increasing the number of vertices in the query graph, and each column shows the increasing of the query time with increasing the number of missing edges.

| Threshold ($\theta$) | $|V(q)|=4$ | $|V(q)|=8$ | $|V(q)|=12$ |
|---------------------|----------|----------|----------|
| 0                   | 22436    | 120964   | 791528   |
| 1                   | 21278    | 65384    | 990676   |
| 2                   | 21901    | 68552    | 814500   |
| 3                   | 21901    | 79364    | 843771   |

![Figure 18: Number of Matches and The Query Time](image)

As a conclusion of our experiment, SAPPER is an effective method to retrieve all approximate matches of the query graph with some missing edges. However, it cannot directly identify the exact matches with $\theta = 0$ in some cases. We also found that the query time is affected by the number of distinct labels in the database graph, the number of vertices in the query graph, and the number of the subgraphs that are enumerated based on the value of the given threshold.
7. CONCLUSION

We implemented SAPPER as a method to retrieve all approximate matches of the query graph with possible missing edges using the pre-generated random spanning trees and the query graph enumeration order. We also studied each process in this method in details with some examples. Finally, in our experiment, we analyzed the performance of SAPPER using a real dataset and identified some case where SAPPER can and cannot retrieve all the exact matches of the query graph. Also we identified the parameters that affect the query time for future work to improve the performance and the scalability of SAPPER to query larger graph.

8. REFERENCES


APPENDIX

A. ALGORITHM DESCRIPTION

Algorithm 1: Vertex Matching

**Input:** Graph database G, Query graph q.
**Output:** a set of matching vertices for every vertex in q.

1. begin
2. for $V_q \in q$ do
3.   for $V_g \in G$ do
4.     if $v_g$'s label = $v_q$'s label then
5.       if $v_g$'s degree $\leq$ $v_q$'s degree then
6.         $V_g$ is a match for $v_q$
7.         $M(v_q)$.add $(V_g)$
8.       Else if $N_1(v_q, q)$ labels $\subseteq N_1(v_g, G)$ then
9.         $M(v_q)$.add $(V_g)$
10.      Else
11.        $V_g$ is not a match for $v_q$

Algorithm 2: Generating Random Spanning Tree

**Input:** The query graph q.
**Output:** Random Spanning Tree of q.

1. begin
2.   Construct transition matrix $P$ from q.
3.   Vertex set $S$ ← $\phi$, edge list $E$ ← $\phi$.
4.   Randomly select a vertex $X_0$ of q.
5.   $S$ ← $S$ + $X_0$
6.   $v$ ← $X_0$.
7.   while $S \leq |V(q)|$ do
8.     Randomly select vertex $w$ by $P$, $evw$ exists.
9.     if $! (w \in S)$ then
10.    $E$ ← $E + evw$
11.   $S$ ← $S + w$
12.  $v$ ← $w$
13. Output the graph composed of edge list $E$.

Algorithm 3: Tree Matching

**Input:** Set of all the spanning trees $T$.
**Output:** Set of matches vertices for each tree.

1. begin
2. for $T_i \in T$ do
3.   for $v_i \in T_i$ based on DFS vertex order do
4.     Get Vertex Matches $M(v)$
5.     Find Connected Vertex to $v_i$ in the tree
6.     if $N_1(v_i, G)$ First neighbors
7.       $u$ ∈ $M(V, T_i)$
8.     else
9.       Remove $v_i$ from $M(v, T_i)$

Algorithm 4: Query Graph Enumeration Order.

**Input:** Query graph edge List = $(v_1 : v_2), (v_2 : v_3), (\forall (v_i : v_j) \in q)$, DFS order of the query graph q $Q$-DFS= $v_1, v_3, \ldots, (\forall (v_i \in q)$). Candidate set $M(v_i)(\forall v_i \in q)$
**Output:** lexicographical order of each edge.

1. begin
2. for $V_q \in q$ based on DFS order do
3.   Find $M(V_q)=v_1, v_2, \ldots$
4. for $U_q$ vertex that connect to $V_q \in q$ do
5.   Get $U_q$ candidate set $M(U_q)$
6. for $u_i \in M(U_q)$ do
7.   if $u_i \cap u_i \in M(V_q)'s$ direct neighbors then
8.     // if there is a match,
9.     then there is an edge
10.   if $(v_i : u_i)'s$ label = $(V_q : U_q)'s$ label then
11.      $EdgeMatches ++$
12.     else
13.      No match $(v_i: u_i)$ in G.
14.      $Edge-Occurrences-List.add((v_i : u_i), EdgeMatches)$
15. Sort the edges in $Edge-Occurrences-List$ smallest to largest based on the number of the matches.
16. Assign a lexicographical order to each edge based on its order in the list.
17. Return The Lexicographical Order;
Algorithm 5: Find Exact Edge's Matches in G

Input: Tree-Matches Results, which include matches by Vertex, Current SubgraphQuery
Output: set of matches for each edge in the subgraph Query

1 begin
2 for each TreeMatches(i) ∈ TreeMatches do
3   for each Edgeei ∈ SubgraphQuery do
4     Vs, ei’s Edge Source.
5     Vt, ei’s Edge Target.
6     // Compare Vs’s first neighbors matches with Tree-Matches(Vt’s)’s matches
7     if Vs’s first neighbors in T ∩ Tree-Matches(Vt’s) then
8       // There is an edge between Vs and Vt in G
9       if ei’s label equal (vs : vt) label
10      then
11         (vs’ smatch : vt’ smatch) is a match for ei’s matches.
12   Return Edge’s Matches set

Algorithm 6: Search for Matches for Undiscovered Edges

Input: prefix’s-Matches, that are matches by edge, currentSequence, Tree-Matches, which are matches by Vertex, SubgraphQuery
Output: All approximate matches of g(s)’s edges that have not been discovered

1 begin
2   Undiscovered Edges = CurrentSequence’s edges – Prefix’s edges
3   for each ei ∈ UndiscoveredEdges do
4     Vq1 ei’s vertex source.
5     Vq2 ei’s edge target.
6     // Iterate over the tree vertex matches of the prefix
7     for each match’s Set ∈ prefixesMatches do
8       Retrieve Vq1 and Vq2 matches.
9       Find the connected edges to Vq1 and Vq2.
10      // Compare Vq1’s matches in Tree-Matches With the prefix’s-Matches of the connected edges ej to Vq1
11     if prefix’s-Matches (ej) contains some of the Tree-Matches (Vq1) then
12       if prefix’s-Matches (el) contains some of the Tree-Matches (Vq2) then
13         if (Vq1’s smatch : Vq2’s smatch) is an edge ∈ G
14           then
15             (Vq1’s match : Vq2’s) ∈ ei’s matches

Algorithm 7: Map The Final Matches

Input: VertexOrder, sequence, subgraphQuery, EdgeMatches
Output: list of all approximate matches of the subgraph query.

1 begin
2   for Vertexi ∈ VertexOrder do
3     // ADJ is a list to represents vertices connectivity in q
4     connectedVertices = ADJ(Vertexi)
5     FirstLoop:
6       for Vi ∈ connectedVertices do
7         // Return the edge number
8         EdgeOrder = GetEdgeNumber (Vertexi, Vi)
9         If (EdgeOrder! = Null )
10          M(EdgeOrder) = EdgeMatches.get(EdgeOrder)
11          Innerloop:
12            for ej ∈ M(EdgeOrder) do
13               if InitialSolution.isEmpty() then
14                 InitialSolution.put(EdgeOrder, ej)
15                 EdgesMatched.add(ej)
16                 Break Innerloop:
17               if (j == M(EdgeOrder).size() − 1)
18                 InitialSolution.remove(EdgeOrder, ej)
19                 Break Firstloop:
20               if EdgesMatched.size() = 2 then
21                 CartesianProduct(EdgesMatched(e1)’s matches, EdgesMatched(e2)’s matches)
22                 // Group each two matches together if there is reference
23                 use CartesianProduct to map the rest of vertices.
Algorithm 8: SAPPER

**Input:** database graph G, query graph q, threshold θ.

**Output:** list of all approximate matches of q.

1 begin
2 Edge list \( EL \leftarrow 1,\ldots,el, \forall e_i \in q \), \( l \leftarrow |E(q)| \).
3 Sort q’s edges decreasingly by their number of matches in G.
4 Assign lexicographical order for the edges in q using Algorithm (4).
5 \( s \leftarrow e_1\ldots sl\theta, |s| = l - \theta \).
6 End is a subgraph that includes the largest lexicographical order.
7 while \( S \neq \text{End} \) do
8 if IsConnected(\( g(s) \)) then
9 if the subgraph \( s \) corresponding to the longest prefix of \( s \) is not matched yet then
10 if \( s \) corresponding to one spanning tree then
11 Find \( g(s) \) matches using The Spanning Tree Matching Algorithm (3)
12 Find Best Vertex’s Search Order.
13 Map the matches (Subgraph’s Matches, Search Order)
14 Algorithm(8)
15 if \( s \) can have more than one spanning tree then
16 Find Random spanning tree (\( g(s) \)) Algorithm(2)
17 Find \( g(s) \) matches using The Spanning Tree Matching Algorithm (3)
18 Find Best Vertex’s Search Order.
19 Map the matches (Subgraph’s Matches, Search Order)
20 Algorithm(8)
21 Else
22 // The longest prefix of the subgraph have been discovered
23 Search for Undiscovered Edges in prefix’s matches, Algorithm(7)
24 if \( g(s) \) has no match then
25 \( s \leftarrow \text{get a new sequence that does not contain } g(s) \text{ as prefix.} \)
26 Else \( s \leftarrow \text{add one edge to } g(s) \)
27 Else
28 // if \( g(s) \) is not connected
29 continue