Structure- and Path-based Graph Queries

Specialization Report, 2014

Andreas Gruhler
12-708-038

1. INTRODUCTION

The goal of querying a database is to find all graphs, in which the query appears as a subgraph [1, 2]. The correct result of query q(figure 2) on database D(figure 1) should be g_4. To retrieve results fast, an *index* has to be built first (section 2). Afterwards the database is *searched* for the specific characteristics of the query using the index (section 3). A candidate query answer set is retrieved. This result has to be *verified*, to ensure that false positives are removed from the set (section 4).

The described high level approach is proposed by numerous authors. This report holds a comparison between the two indexing methods *GraphGrep* [2] and *gIndex* [1]. Experiments of Yan et al. already led to the conclusion that gIndex performs better and the index size is smaller in comparison to GraphGrep [1].

1.1 Notation

Throughout this report, the following notation will be used:

Symbol	Description
D	A graph database
g	A graph inside a database
q	A query graph
f	A fragment/subgraph
D_f	A set of graphs in D containing fragment f as subgraph, $D_f = \{g_i \mid f \subseteq g_i, g_i \in D\}$
$\lvert D_f \rvert$	Support of fragment f in database D
C_q	The candidate query answer set retrieved using an index
γ_{min}	Minimum discriminative ratio
$minSup$	Minimum support
$dfs(g)$	Canonical label (minimum dfs code) of g

The examples in this report make use of database D(figure 1) and query graph q(figure 2).

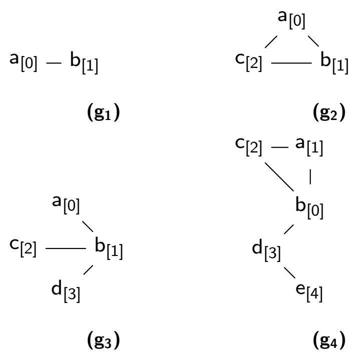

Figure 1: A sample database $D = \{g_1, g_2, g_3, g_4\}$. Subscripted numbers in brackets are only used for the GraphGrep approach.

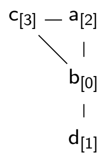

Figure 2: A sample query q. Subscripted numbers in brackets are only used for the GraphGrep approach.

2. INDEX CONSTRUCTION

The two methods use different characteristics of graphs to build up an index. GraphGrep indexes paths up to a given length [2] whereas the gIndex stores frequent and discriminative subgraphs [1]. These concepts will be elaborated in the next two subsections.

Especially in the improved information quality of graphs, Yan et al. see a major advantage in comparison to paths [1]. The authors of gIndex see two main disadvantages in their choice of storing subgraphs [1]:

1. Graphs are more difficult to manipulate than paths.
2. Fully connected graphs may contain a very large amount of subgraphs. Considering the index size, not all of them can be stored.

2.1 GraphGrep Index Construction

When using GraphGrep [2], the index stores for each graph in a database all the paths of length 1(one node) up to a fixed length l_p by the sequence of labels on the nodes in the path, and the sequence of ids on the nodes in the path (see figure 3 as example for g_4). In addition to the paths, a hash table (called *fingerprint*) which keeps track of the occurrences of all paths needs to be stored for each graph (see figure 4) [2].

$$\begin{aligned}
a =& \{(1)\}\ ac = \{(1,2)\}\ acb = \{(1,2,0)\}\\
ab =& \{(1,0)\}\ abc = \{(1,0,2)\}\ abd = \{(1,0,3)\}\\
b =& \{(0)\}\ ba = \{(0,1)\}\ bac = \{(0,1,2)\}\\
bc =& \{(0,2)\}\ bca = \{(0,2,1)\}\\
bd =& \{(0,3)\}\ bde = \{(0,3,4)\}\\
c =& \{(2)\}\ ca = \{(2,1)\}\ cab = \{(2,1,0)\}\\
cb =& \{(2,0)\}\ cba = \{(2,0,1)\}\ cbd = \{(2,0,3)\}\\
d =& \{(3)\}\ de = \{(3,4)\}\\
db =& \{(3,0)\}\ dba = \{(3,0,1)\}\ dbc = \{(3,0,2)\}\\
e =& \{(4)\}\ ed = \{(4,3)\}\ edb = \{(4,3,0)\}
\end{aligned}$$

Figure 3: All paths up to length $l_p = 3$ in g_4.

Key	g_1	g_2	g_3	g_4
$h(a)$	1	1	1	1
$h(ac)$	0	1	0	1
$h(acb)$	0	1	0	1
$h(ab)$	1	1	1	1
$h(abc)$	0	1	1	1
$h(abd)$	0	0	1	1
$h(b)$	1	1	1	1
$h(ba)$	1	1	1	1
$h(bac)$	0	1	0	1
$h(bc)$	0	1	1	1
$h(bca)$	0	1	0	1
$h(bd)$	0	0	1	1
$h(bde)$	0	0	0	1
$h(c)$	0	1	1	1
$h(ca)$	0	1	0	1
$h(cab)$	0	1	0	1
$h(cb)$	0	1	1	1
$h(cba)$	0	1	1	1
$h(cbd)$	0	0	1	1
$h(d)$	0	0	1	1
$h(db)$	0	0	1	1
$h(dba)$	0	0	1	1
$h(dbc)$	0	0	1	1
$h(de)$	0	0	0	1
$h(e)$	0	0	0	1
$h(ed)$	0	0	0	1
$h(edb)$	0	0	0	1

Figure 4: The fingerprint of database D.

According to Sasha et al. [2], the GraphGrep time complexity for index construction is $O(\sum_i^{|D|}(n_i m_i^{l_p}))$ and space complexity is $O(\sum_i^{|D|}(l_p n_i m_i^{l_p}))$. n_i is the number of nodes in a graph $g_i \in D$ and m_i the maximum degree in this graph.

2.2 gIndex Construction

The gIndex is a structure based indexing approach, where frequent and discriminative fragments are stored [1]. Figure 5 shows all the subgraphs of D and q, also called fragments [1].

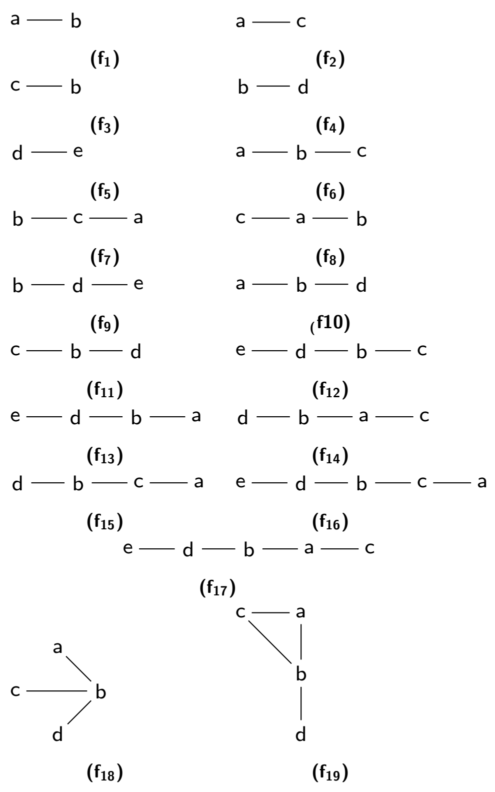

Figure 5: Subgraphs of D and q.

Every fragment f has a level of support($|D_f|$) in D, which represents the number of graphs in D, in which f can be found as a subgraph [1]. A fragment f is called frequent fragment, if $|D_f| \geq minSup$ [1]. Figure 6 shows the support levels and the discriminative ratio γ [1] for the fragments in figure 5. The discriminative ratio is used by Yan et al. to further decrease index size, without loosing important fragments. According to Yan et al. γ is computed as follows [1]:

$$\gamma = \frac{|\bigcap_i D_{f_i}|}{|D_f|}$$

The counter corresponds to the number of graphs supported by all subfragments $f_i \subset f$.

A fragment f is indexed if $|D_f| \geq minSup \wedge |\bigcap_i D_{f_i}|/|D_f| \geq \gamma_{min}$ [1]. Figure 7 shows the gIndex tree with $\gamma_{min} = 1$ and minimum support 2, where each indexed fragment is represented by the minimum dfs code $dfs(g)$(also called *canonical label* [3, 4]) and the index is modelled as a prefix tree of dfs codes. White nodes are intermediate fragments [1], which are not discriminative fragments. Colored nodes are the frequent and discriminative fragments. Yan et al. store the tree as hash table (see figure 8) [1].

f_i	D_{f_i}	$\|D_{f_i}\|$	γ
f_1	$\{g_1 - g_4\}$	4	$\frac{4}{4} = 1$
f_2	$\{g_2, g_4\}$	2	$\frac{3}{2} = 1.5$
f_3	$\{g_2 - g_4\}$	3	$\frac{3}{3} = 1$
f_4	$\{g_3, g_4\}$	2	$\frac{2}{2} = 1$
f_5	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_6	$\{g_2 - g_4\}$	3	$\frac{3}{3} = 1$
f_7	$\{g_2, g_4\}$	2	$\frac{2}{2} = 1$
f_8	$\{g_2, g_4\}$	2	$\frac{2}{2} = 1$
f_9	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{10}	$\{g_3, g_4\}$	2	$\frac{2}{2} = 1$
f_{11}	$\{g_3, g_4\}$	2	$\frac{2}{2} = 1$
f_{12}	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{13}	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{14}	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{15}	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{16}	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{17}	$\{g_4\}$	1	$\frac{1}{1} = 1$
f_{18}	$\{g_3, g_4\}$	2	$\frac{2}{2} = 1$
f_{19}	$\{g_4\}$	1	$\frac{1}{1} = 1$

Figure 6: Support and γ of fragments f_i in D.

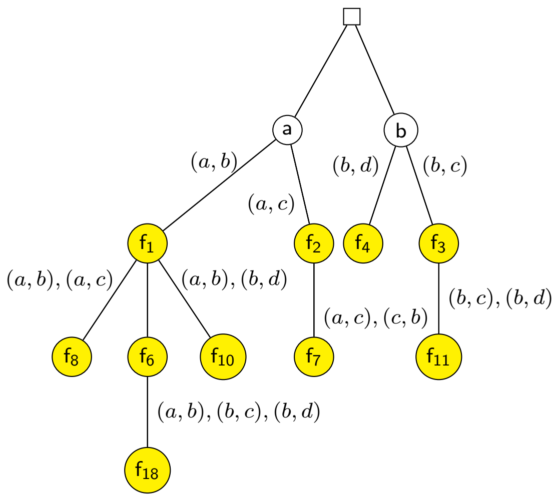

Figure 7: The gIndex tree of D with minimum support 2 and $\gamma_{min} = 1$.

In the example, the gIndex(figure 8) uses less space than GraphGrep(figure 3 and 4). GraphGrep would even use more space than visible on the figures, because a similar path representation like figure 3 has to be stored for each graph in D. This space advantage of gIndex can reduce search time. Yan et al. claim the gIndex is smaller by a factor of 10 compared to a path based index.

The gIndex authors do not give an estimate on index construction time, but clearly it depends on the chosen parameters(minimum support, γ_{min}). Techniques to find frequent fragments efficiently are available [1].

Key	Links to $g \in D$
$h(a)$	$\emptyset$
$h(b)$	$\emptyset$
$h(dfs(f_1))$	$\{g_1 - g_4\}$
$h(dfs(f_2))$	$\{g_2, g_4\}$
$h(dfs(f_3))$	$\{g_2 - g_4\}$
$h(dfs(f_4))$	$\{g_3, g_4\}$
$h(dfs(f_6))$	$\{g_2 - g_4\}$
$h(dfs(f_7))$	$\{g_2, g_4\}$
$h(dfs(f_8))$	$\{g_3, g_4\}$
$h(dfs(f_{10}))$	$\{g_3, g_4\}$
$h(dfs(f_{11}))$	$\{g_3, g_4\}$
$h(dfs(f_{18}))$	$\{g_3, g_4\}$

Figure 8: The gIndex tree from figure 7 as hash table (the gIndex as it is stored in memory). The frequent discriminative nodes link to their supporting graphs. Intermediate nodes have no values.

There even exists the possibility to only look for frequent and discriminative fragments in a sample of D [1]. The rest of the index can then be updated incrementally, as long as the new graphs do not change the frequency and the γ of the already indexed fragments [1]. According to Yan et al., this is rarely the case if the data is from the same dataset.

GraphGrep on the other hand does not come with such an incremental building strategy.

3. SEARCH

As input for an index lookup, a constructed index and the query graph from last section is needed. The goal of this search phase in a graph query is to retrieve a set of candidate graphs. The set may contain false positives(wrong results), which will be removed in the verification phase(section 4).

Yan et al. already pointed out, that for graph queries where structure offers more valuable information than only its paths, a path based approach like GraphGrep does not provide the necessary pruning power [1]. A similar demonstrative example will be given at the end of subsection 3.1 by modifying the example database D from figure 1. When applying the gIndex approach to this modified database in the end of subsection 3.2, we will see that it really performs better on such graphs.

3.1 GraphGrep Search

In GraphGrep [2], a query fingerprint(figure 9) gets compared with the fingerprint of D(figure 4). Graphs in D with less occurrences of a path (compared to q) are pruned from the result. Those graphs are g_1, g_2, g_3. The remaining candidate query set is $\{g_4\}$. The cost/time for filtering/search depends on the size of the query fingerprint.

Key	q
$h(a)$	1
$h(b)$	1
$h(c)$	1
$h(d)$	1
$h(ab)$	1
$h(ac)$	1
$h(ba)$	1
$h(bc)$	1
$h(bd)$	1
$h(ca)$	1
$h(cb)$	1
$h(db)$	1
$h(abc)$	1
$h(abd)$	1
$h(acb)$	1
$h(bac)$	1
$h(bca)$	1
$h(cab)$	1
$h(cba)$	1
$h(cbd)$	1
$h(dba)$	1
$h(dbc)$	1

Figure 9: The fingerprint of query q.

If we modify the example database D and the query q from figure 1 and 2 and give all nodes the same label c, we can see that GraphGrep performs differently. Still having $l_p = 3$, the fingerprint of D would look like this:

Key	g_1	g_2	g_3	g_4
$h(c)$	1	1	1	1
$h(cc)$	1	1	1	1
$h(ccc)$	0	1	1	1

The fingerprint of query q would be:

Key	q
$h(c)$	1
$h(cc)$	1
$h(ccc)$	1

The candidate set is $\{g_2, g_3, g_4\}$. g_2 and g_3 cannot be pruned in this case, because the fingerprint does not capture the structure of the graph. This problem was already discovered by Yan et al. and was the motivation behind structure based indexes [1].

3.2 gIndex Search

To search the gIndex tree, Yan et al. use apriori pruning and maximum discriminative fragments [1].

Apriori pruning means, that supergraphs of fragments which are not found in the index do not need to be looked up in the index anymore [1]. The subgraphs of q are $\{f_1 - f_4, f_6 - f_8, f_{10}, f_{11}, f_{14}, f_{15}, f_{18}, f_{19}\}$. For each subgraph, Yan et al. look into the hash table(figure 8), starting with the smallest fragment. Because f_{14} and f_{15} are not in the table, f_{19} can be pruned. This is what Yan et al. call apriori pruning [1] and it is easy to prune because the gIndex tree(figure 7) also stores intermediate nodes.

The maximum discriminative fragments are the deepest discriminative(colored) nodes in the gIndex tree, which are still contained in the query [1]. The maximum discriminative fragments in the remaining set $\{f_1 - f_4, f_6 - f_8, f_{10}, f_{11}, f_{18}\}$ are $\{f_8, f_{18}, f_{10}, f_7, f_4, f_{11}\}$. Fragments $\{f_1 - f_3, f_6\}$ are being pruned because there exist deeper discriminative nodes which are still contained by the query.

The candidate query set is $\{g_4\}$. This is equal to the intersection of all links of the maximum discriminative fragments, which are the colored leaf nodes of the tree (see figure 7 and 8).

In the example, GraphGrep does 22 index look-ups(fingerprint size of q), whereas gIndex only needs to do 6($\{f_8, f_{18}, f_{10}, f_7, f_4, f_{11}\}$) thanks to apriori pruning and the use of maximum discriminative fragments.

If we apply the label change as in subsection 3.1 (we change all the labels in D and q to be the same), the subgraphs of D and q would look like in figure 10.

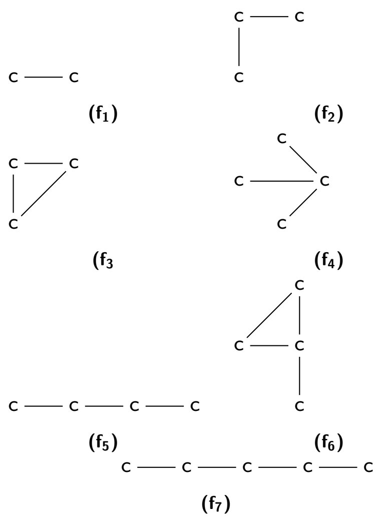

Figure 10: Subgraphs of D and q if all nodes of D and q would have the same label c.

Support level and γ of those new fragments from figure 10 would be as follows:

f_i	D_{f_i}	$\lvert D_{f_i}\rvert$	γ
f_1	$\{g_1 - g_4\}$	4	1
f_2	$\{g_2 - g_4\}$	3	$\frac{4}{3}$
f_3	$\{g_2, g_4\}$	2	$\frac{3}{2}$
f_4	$\{g_3, g_4\}$	2	$\frac{3}{2}$
f_5	$\{g_4\}$	1	3
f_6	$\{g_4\}$	1	1
f_7	$\{g_4\}$	1	1

If we decide not to change the minimum support of 2 and $\gamma_{min} = 1$, the index would consist of fragments $\{f_1 - f_4\}$ (the fragment number refers now to figure 10). The gindex tree would look like the tree in figure 11 and the hash table as in figure 12.

The modified query q with all the nodes having the same label would then have fragments $\{f_1 - f_6\}$ as subgraphs. Since f_5 is not indexed, the apriori pruning will remove supergraph f_6. In this case the candidate query answer set calculated using maximum discriminative fragments would be $\bigcap_{i=3}^{4} D_{f_i} = \{g_4\}$.

This is a smaller answer set than in section 3.1. According to Yan et al. this is better, because the overall query consists of the following components [1]:

$$T_{search} + |C_q| * T_{verification}$$

T_{search} is the time to search, $T_{verification}$ the time to verify the result and $|C_q|$ the size of the candidate answer set.

4. VERIFICATION

4.1 GraphGrep Verification

GraphGrep creates a DFS-tree of q and defines patterns of length l_q or less [2]. The patterns of q with $l_q = 3$ are $c^*a\underline{b}$, $\underline{b}c^*$ and $\underline{b}d$. Overlapping labels are marked with underscore and asterix. The paths in g_4(the remaining graph after filtering) matching these labels are:

$$\begin{aligned} cab &= \{(2,1,0)\} \\ bc &= \{(0,2)\} \\ bd &= \{(0,3)\} \end{aligned}$$

Afterwards, Sasha et al. [2] combine the paths, if allowed by a pattern, to find the exact location of q in g_4:

$$\begin{aligned} cabc &= \{((2,1,0),(0,2))\} \\ cabd &= \{((2,1,0),(0,3))\} \end{aligned}$$

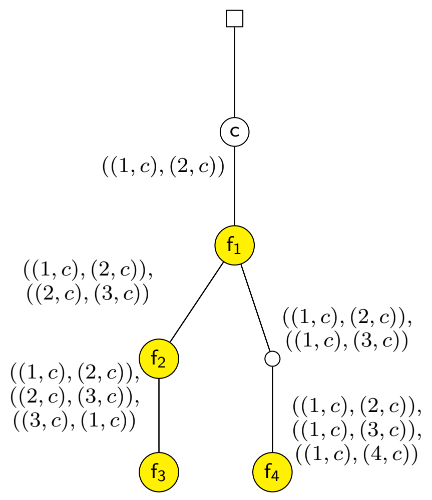
c
((1,c),(2,c))
f1
((1,c),(2,c)), ((2,c),(3,c))
((1,c),(2,c)), ((1,c),(3,c))
f2
((1,c),(2,c)), ((2,c),(3,c)), ((3,c),(1,c))
((1,c),(2,c)), ((1,c),(3,c)), ((1,c),(4,c))
f3
f4

Figure 11: The gIndex tree with minimum support 2 and $\gamma_{min} = 1$ for database D if all nodes of every graph in D would have the same label c.

Key	Links to $g \in D$
$h(c)$	$\emptyset$
$h(((1,c),(2,c)),((1,c),(3,c)))$	$\emptyset$
$h(dfs(f_1))$	$\{g_1 - g_4\}$
$h(dfs(f_2))$	$\{g_2 - g_4\}$
$h(dfs(f_3))$	$\{g_2, g_4\}$
$h(dfs(f_4))$	$\{g_3, g_4\}$

Figure 12: The gIndex tree from figure 11 as hash table.

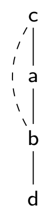

Figure 13: A DFS-tree of q.

According to Sasha et al. [2], the pattern matching has complexity $O(\sum_i^{|D_f|}(\tilde{n}_i m_i^{l_p})^p)$. p is the number of patterns, $\tilde{n}$ the maximum number of nodes having the same label and $|D_f|$ is the size of the database after filtering [2].

4.2 gIndex Verification

To verify the candidate query set, gIndex does subgraph isomorphism tests [1]. This ensures that the remaining graphs share common structure with the query graph [1].

5. REFERENCES

[1] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In SIGMOD '04, pages 335-346.

[2] D. Shasha, J.T-L Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In PODS '02, pages 39-52.

[3] Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In ICDM '02, pages 721-724.

[4] Xifeng Yan and Jiawei Han. CloseGraph: Mining closed frequent graph patterns. In KDD '03, pages 286-295.