gSpan: Graph-Based Substructure Pattern Mining

Xifeng Yan Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
{xyan, hanj}@uiuc.edu

Abstract

We investigate new approaches for frequent graph-based pattern mining in graph datasets and propose a novel algorithm called gSpan (graph-based Substructure pattern mining), which discovers frequent substructures without candidate generation. gSpan builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, gSpan adopts the depth-first search strategy to mine frequent connected subgraphs efficiently. Our performance study shows that gSpan substantially outperforms previous algorithms, sometimes by an order of magnitude.

1. Introduction

Frequent substructure pattern mining has been an emerging data mining problem with many scientific and commercial applications. As a general data structure, labeled graph can be used to model much complicated substructure patterns among data. Given a graph dataset, $D = \{G_0, G_1, ..., G_n\}$, $support(g)$ denotes the number of graphs (in D) in which g is a subgraph. The problem of *frequent subgraph mining* is to find any subgraph g s.t. $support(g) \geqslant minSup$ (a minimum support threshold). To reduce the complexity of the problem (meanwhile considering the connectivity property of hidden structures in most situations), only frequent connected subgraphs are studied in this paper.

The kernel of frequent subgraph mining is subgraph isomorphism test. Lots of well-known pair-wise isomorphism testing algorithms were developed. However, the frequent subgraph mining problem was not explored well. Recently, Inokuchi et al. [4] proposed an Apriori-based algorithm, called AGM, to discover all frequent (both connected and disconnected) substructures. Kuramochi and Karypis [5] further developed the idea using adjacent representation of graph and an edge-growing strategy. Their algorithm, called FSG, is able to find all frequent connected subgraphs from a chemical compound dataset in 10 minutes with 6.5% minimum support. For the same dataset, our novel algorithm can complete the same task in 10 seconds.

AGM and FSG both take advantage of the Apriori level-wise approach [1]. In the context of frequent subgraph mining, the Apriori-like algorithms meet two challenges: (1) candidate generation: the generation of size $(k+1)$ subgraph candidates from size k frequent subgraphs is more complicated and costly than that of itemsets; and (2) pruning false positives: subgraph isomorphism test is an NP-complete problem, thus pruning false positives is costly.

Contribution. In this paper, we develop *gSpan*, which targets to reduce or avoid the significant costs mentioned above. If the entire graph dataset can fit in main memory, *gSpan* can be applied directly; otherwise, one can first perform graph-based data projection as in [6], and then apply *gSpan*. To the best of our knowledge, *gSpan* is the first algorithm that explores depth-first search (DFS) in frequent subgraph mining. Two techniques, *DFS lexicographic order* and *minimum DFS code*, are introduced here, which form a novel canonical labeling system to support DFS search. *gSpan* discovers all the frequent subgraphs without candidate generation and false positives pruning. It combines the growing and checking of frequent subgraphs into one procedure, thus accelerates the mining process.

2. DFS Lexicographic Order

This section introduces several techniques developed in *gSpan*, including *mapping each graph to a DFS code (a sequence), building a novel lexicographic ordering among these codes,* and *constructing a search tree based on this lexicographic order.*

DFS Subscripting. When performing a depth-first search [3] in a graph, we construct a DFS tree. One graph can have several different DFS trees. For example, graphs in Fig. 1(b)-(d) are isomorphic to that in Fig. 1(a). The thickened edges in Fig. 1(b)-(d) represent three different DFS trees for the graph in Fig. 1(a). The depth-first discovery of the vertices forms a linear order. We use subscripts to label this

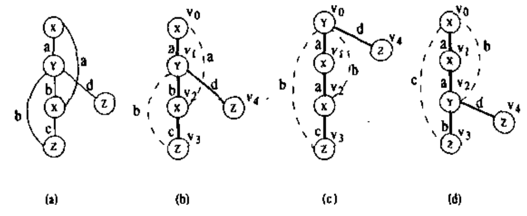

Figure 1. Depth-First Search Tree

edge	(Fig 1b) α	(Fig 1c) β	(Fig 1d) γ
0	$(0,1,X,a,Y)$	$(0,1,Y,a,X)$	$(0,1,X,a,X)$
1	$(1,2,Y,b,X)$	$(1,2,X,a,X)$	$(1,2,X,a,Y)$
2	$(2,0,X,a,X)$	$(2,0,X,b,Y)$	$(2,0,Y,b,X)$
3	$(2,3,X,c,Z)$	$(2,3,X,c,Z)$	$(2,3,Y,b,Z)$
4	$(3,1,Z,b,Y)$	$(3,0,Z,b,Y)$	$(3,0,Z,c,X)$
5	$(1,4,Y,d,Z)$	$(0,4,Y,d,Z)$	$(2,4,Y,d,Z)$

Table 1. DFS codes for Fig. 1(b)-(d)

order according to their discovery time [3]. $i < j$ means v_i is discovered before v_j. We call v_0 the *root* and v_n the *rightmost vertex*. The straight path from v_0 to v_n is named the *rightmost path*. In Fig. 1(b)-(d), three different subscriptings are generated for the graph in Fig. 1(a). The right most path is (v_0, v_1, v_4) in Fig. 1(b), (v_0, v_4) in Fig. 1(c), and (v_0, v_1, v_2, v_4) in Fig. 1(d). We denote such subscripted G as G_T.

Forward Edge and Backward Edge. Given G_T, the forward edge (*tree edge* [3]) set contains all the edges in the DFS tree, and the backward edge (*back edge* [3]) set contains all the edges which are not in the DFS tree. For simplicity, (i, j) is an ordered pair to represent an edge. If $i < j$, it is a forward edge; otherwise, a backward edge. A linear order, $\prec_T$ is built among all the edges in G by the following rules (assume $e_1 = (i_1, j_1), e_2 = (i_2, j_2)$): (i) if $i_1 = i_2$ and $j_1 < j_2$, $e_1 \prec_T e_2$; (ii) if $i_1 < j_1$ and $j_1 = i_2$, $e_1 \prec_T e_2$; and (iii) if $e_1 \prec_T e_2$ and $e_2 \prec_T e_3$, $e_1 \prec_T e_3$.

Definition 1 (DFS Code) *Given a DFS tree T for a graph G, an edge sequence (e_i) can be constructed based on $\prec_T$, such that $e_i \prec_T e_{i+1}$, where $i = 0, \ldots, |E| - 1$. (e_i) is called a DFS code, denoted as code(G, T).*

For simplicity, an edge can be presented by a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$, where l_i and l_j are the labels of v_i and v_j respectively and $l_{(i,j)}$ is the label of the edge between them. For example, (v_0, v_1) in Fig. 1(b) is represented by $(0, 1, X, a, Y)$. Table 1 shows the corresponding DFS codes for Fig. 1(b), 1(c), and 1(d).

Definition 2 (DFS Lexicographic Order) *Suppose $Z = \{code(G,T) \mid T$ is a DFS tree of $G\}$, i.e., Z is a set containing all DFS codes for all the connected labeled graphs. Suppose there is a linear order ($\prec_L$) in the label set (L), then the lexicographic combination of $\prec_T$ and $\prec_L$ is a linear order ($\prec_e$) on the set $E_T \times L \times L \times L$. For further details see [7].* ***DFS Lexicographic Order*** *is a linear order defined as follows. If $\alpha = code(G_\alpha, T_\alpha) = (a_0, a_1, ..., a_m)$ and $\beta = code(G_\beta, T_\beta) = (b_0, b_1, ..., b_n), \alpha, \beta \in Z$, then $\alpha \leqslant \beta$ iff either of the following is true.*

$$(i) \quad \exists t, 0 \leqslant t \leqslant min(m,n), a_k = b_k \ for \ k < t, a_t \prec_e b_t$$

$$(ii) \qquad a_k = b_k \ for \ 0 \leqslant k \leqslant m, \ and \ n \geqslant m.$$

For the graph in Fig. 1 (a), there exist tens of different DFS codes. Three of them, which are based on the DFS trees in Fig. 1(b)-(d) are listed in Table 1. According to DFS lexicographic order, $\gamma \prec \alpha \prec \beta$.

Definition 3 (Minimum DFS Code) *Given a graph G, $Z(G) = \{code(G,T) \mid T$ is a DFS tree of $G\}$, based on DFS lexicographic order, the minimum one, $min(Z(G))$, is called* ***Minimum DFS Code*** *of G. It is also a canonical label of G.*

Theorem 1 *Given two graphs G and G', G is isomorphic to G' if and only if $min(G) = min(G')$. (proof omitted)*

Thus the problem of mining frequent connected subgraphs is equivalent to mining their corresponding minimum DFS codes. This problem turns to be a sequential pattern mining problem with slight difference, which conceptually can be solved by existing sequential pattern mining algorithms.

Given a DFS code $\alpha = (a_0, a_1, ..., a_m)$, any valid DFS code $\beta = (a_0, a_1, ..., a_m, b)$, is called α's **child**, and α is called β's **parent**. In fact, to construct a valid DFS code, b must be an edge which only grows from the vertices on the rightmost path. In Fig. 2, the graph shown in 2(a) has several potential children with one edge growth, which are shown in 2(b)-(f) (assume the darkened vertices constitute the rightmost path). Among them, 2(b), 2(c), and 2(d) grow from the rightmost vertex while 2(e) and 2(f) grow from other vertices on the rightmost path. 2(b.0)-(b.3) are children of 2(b), and 2(e.0)-(e.2) are children of 2(e). Backward edges can only grow from the rightmost vertex while forward edges can grow from vertices on the rightmost path. This restriction is similar to TreeMinerV's equivalence class extension [8] and FREQT's rightmost expansion [2] in frequent tree discovery. The enumeration order of these children is enhanced by the DFS lexicographic order, i.e., it should be in the order of 2(b), 2(c), 2(d), 2(e), and 2(f).

Definition 4 (DFS Code Tree) *In a DFS Code Tree, each node represents a DFS code, the relation between parent and child node complies with the parent-child relation described above. The relation among siblings is consistent*

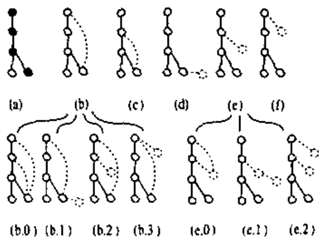

Figure 2. DFS Code/Graph Growth

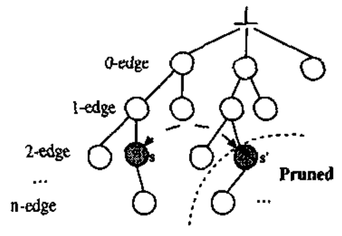

Figure 3. A Search Space: DFS Code Tree

with the DFS lexicographic order. That is, the pre-order search of DFS Code Tree follows the DFS lexicographic order.

Given a label set L, a DFS Code Tree should contain an infinite number of graphs. Since we only consider frequent subgraphs in a finite dataset, the size of a DFS Code Tree is finite. Fig. 3 shows a DFS Code Tree, the n_{th} level nodes contain DFS codes of $(n-1)$-edge graphs. Through depth-first search of the code tree, all the minimum DFS codes of frequent subgraphs can be discovered. That is, all the frequent subgraphs can be discovered in this way. We should mention that if in Fig. 3 the darken nodes contain the same graph but different DFS codes, then s' is not the minimum code (proved in [7]). Therefore, the whole sub-branch of s' can be pruned since it will not contain any minimum DFS code.

3. The *gSpan* Algorithm

We formulate the *gSpan* algorithm in this section. *gSpan* uses a sparse adjacency list representation to store graphs. Algorithm 1 outlines the pseudo-code of the framework, which is self-explanatory (Note that $\mathbb{D}$ represents the graph dataset, $\mathbb{S}$ contains the mining result).

Assume we have a label set $\{A, B, C, \ldots\}$ for vertices, and $\{a, b, c, \ldots\}$ for edges. In Algorithm 1 line 7-12, the first round will discover all the frequent subgraphs containing an edge $A \overset{a}{-} A$. The second round will discover all the frequent subgraphs containing $A \overset{a}{-} B$, but not any $A \overset{a}{-} A$. This procedure repeats until all the frequent subgraphs are discovered. The database is shrunk when this procedure continues (Algorithm 1-line 10) and when the subgraph turns to be larger (Subprocedure 1-line 8, only graphs which contains this subgraph are considered. D_s means the set of graphs in which s is a subgraph). Subgraph_Mining is recursively called to grow the graphs and find all their frequent descendants. Subgraph_Mining stops searching either when the support of a graph is less than $minSup$, or its code is not a minimum code, which means this graph and all its descendants have been generated and discovered before (see [7]).

Algorithm 1 GraphSet_Projection($\mathbb{D}$,$\mathbb{S}$).

```
1: sort the labels in $\mathbb{D}$ by their frequency;
2: remove infrequent vertices and edges;
3: relabel the remaining vertices and edges;
4: $\mathbb{S}^1 \leftarrow$ all frequent 1-edge graphs in $\mathbb{D}$;
5: sort $\mathbb{S}^1$ in DFS lexicographic order;
6: $\mathbb{S} \leftarrow \mathbb{S}^1$;
7: for each edge $e \in \mathbb{S}^1$ do
8:     initialize $s$ with $e$, set $s.D$ by graphs which contains $e$;
9:     Subgraph_Mining($\mathbb{D}$, $\mathbb{S}$, $s$);
10:    $\mathbb{D} \leftarrow \mathbb{D} - e$;
11:    if $|\mathbb{D}| < minSup$;
12:        break;
```

Subprocedure 1 Subgraph_Mining($\mathbb{D}$, $\mathbb{S}$, s).

```
1: if $s \neq min(s)$
2:     return;
3: $\mathbb{S} \leftarrow \mathbb{S} \cup \{s\}$;
4: enumerate $s$ in each graph in $\mathbb{D}$ and count its children;
5: for each c, c is s' child do
6:     if $support(c) \geq minSup$
7:         $s \leftarrow c$;
8:         Subgraph_Mining($\mathbb{D}_s$, $\mathbb{S}$, $s$);
```

4. Experiments and Performance Study

A comprehensive performance study has been conducted in our experiments on both synthetic and real world datasets. We use a synthetic data generator provided by Kuramochi and Karypis [5]. The real data set we tested is a chemical compound dataset. All the experiments of *gSpan* are done on a 500MHZ Intel Pentium III PC with 448 MB main memory, running Red Hat Linux 6.2. We also implemented our version of FSG which achieves similar perfor-

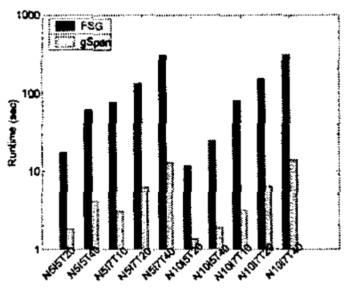

Figure 4. Runtime: Synthetic data

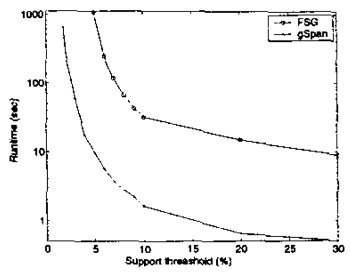

Figure 5. Runtime: Chemical data

mance as that reported in [5]. As shown in Figures 4 and 5, we compare the performance of *gSpan* with FSG [5] if the result is available; otherwise we show our own implementation result based on the same dataset. [5] did the test on a Linux machine with similar configuration.

Synthetic Datasets. The synthetic datasets are generated using a similar procedure described in [1]. Kuramochi et al. [5] applied a simplified procedure in their graph data synthesis. We use their data generator. *gSpan* was tested in various synthetic datasets with different parameters, $|N|$ (the number of possible labels), $|I|$ (the average size of potential frequent subgraphs-kernels), $|T|$ (the average size of graphs in terms of edges) and fixed parameters, $|D| = 10K$ (the total number of graphs generated), $|L| = 200$ (the number of potentially frequent kernels), and $minSup = 0.01 \times |D|$. As shown in Fig. 4, the speed-up is between 6 and 30.

Chemical Compound Dataset. The chemical compound dataset can be retrieved through this URL [1]. The dataset contains 340 chemical compounds, 24 different atoms, 66 atom types, and 4 types of bonds. The dataset is sparse, containing on average 27 vertices per graph and 28 edges per graph. The largest one contains 214 edges and 214 vertices. So the discovered patterns are much like tree, though they do contains some cycles. We use the type of atoms and bonds as labels. The goal is to find the common chemical compound substructures. Fig. 5 illustrates the runtime of *gSpan* and FSG as $minSup$ varies from 2% to 30%. The total memory consumption is less than 100M for any point of *gSpan* plotted in the figure. For FSG, when the $minSup$ is less than 5%, the process is aborted either because the main memory is exhausted or the runtime is too long. Fig. 5 shows *gSpan* achieves better performance by 15-100 times in comparison with FSG.

[1] http://oldwww.comlab.ox.ac.uk/oucl/groups/machlearn/PTE.

5. Conclusions

In this paper, we introduced a new lexicographic ordering system and developed a depth-first search-based mining algorithm *gSpan* for efficient mining of frequent subgraphs in large graph database. Our performance study shows that *gSpan* outperforms FSG by an order of magnitude and is capable to mine large frequent subgraphs in a bigger graph set with lower minimum supports than previous studies.

Acknowledgements. The synthetic data generator is kindly provided by Mr. Michihiro Kuramochi and Professor George Karypis in University of Minnesota. Dr. Pasquale Forggia, at Dipartimento di Informatica e Sistemistica Università di Napoli " Federico II ", provided helpful suggestions about the usage of VFlib graph matching library. We also thank Yanli Tong for her comments.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487–499, Sept. 1994.
[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SIAM SDM'02*, April 2002.
[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001, Second Edition.
[4] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD'00*, pages 13–23, 2000.
[5] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM'01*, pages 313–320, Nov. 2001.
[6] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE'01*, pages 215–224, April 2001.
[7] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. Technical Report UIUCDCS-R-2002-2296, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.
[8] M. J. Zaki. Efficiently mining frequent trees in a forest. In *KDD'02*, July 2002.