Duplicate Candidate Elimination and Fast Support Calculation for Frequent Subgraph Mining

Andrés Gago-Alonso[1,2], Jesús Ariel Carrasco-Ochoa[2], José Eladio Medina-Pagola[1], and José Fco. Martínez-Trinidad[2]

[1] Advanced Technologies Application Center (CENATAV), 7a ♯ 21812 e/ 218 y 222, Rpto. Siboney, Playa, CP: 12200, La Habana, Cuba. {agago,jmedina}@cenatav.co.cu

[2] National Institute of Astrophysics, Optics and Electronics (INAOE), Luis Enrique Erro No. 1, Sta. María Tonantzintla, Puebla, CP: 72840, Mexico. {ariel,fmartine}@inaoep.mx

Abstract. Frequent connected subgraph mining (FCSM) is an interesting task with wide applications in real life. Most of the previous studies are focused on pruning search subspaces or optimizing the subgraph isomorphism (SI) tests. In this paper, a new property to remove all duplicate candidates in FCSM during the enumeration is introduced. Based on this property, a new FCSM algorithm called gdFil is proposed. In our proposal, the candidate space does not contain duplicates; therefore, we can use a fast evaluation strategy for reducing the cost of SI tests without wasting memory resources. Thus, we introduce a data structure to reduce the cost of SI tests. The performance of our algorithm is compared against other reported algorithms.

1 Introduction

Frequent connected subgraph mining (FCSM) in collections of labeled graphs is the process of finding connected subgraphs that occur frequently. Recently, this topic has been an interesting theme in data mining researches with wide applications, including mining substructures from: chemical compound databases, XML documents, citation networks, biological networks, and so forth [3].

The first frequent subgraph miner called AGM was introduced by Inokuchi *et al.* for unconnected graphs [6]. This algorithm was followed by the FSG [8] and AcGM [7] algorithms for mining frequent connected subgraphs.

To avoid overheads of the earlier algorithms, new pattern growth based algorithms such as gSpan [11], MoFa [1], FFSM [5], and Gaston [9] were developed. In all of these approaches, the emergence of duplicate candidates during the mining is one of the major problems. The duplicate candidates are faced by representing subgraphs with a unique code called canonical form (CF) and performing CF tests. However, the computational complexity of a CF test is very high. The DFS code (Depth First Search code) is an example of a promising kind of CF for FCSM [11]. Recently, some properties of the DFS code which are useful for reducing the number of candidates in FCSM were introduced and a new algorithm called gRed based on these properties was described [2].

The subgraph isomorphism (SI) test is another hard subtask in FCSM. The embedding structures (ES) used by MoFa, FFSM, and Gaston avoid this kind of tests but using ES could be a problem if not enough memory is available.

The basic motivation of this work is to develop an integral solution for FCSM facing, at the same time, the two hardest subtasks (CF tests and SI tests). This solution will get better runtimes by removing all duplicate candidates and building the DFSE structure only for useful candidates.

In this paper, we propose a novel cut property of the DFS code which allows to remove all duplicate candidates in FCSM during the candidate enumeration. Besides, we introduce an ES, called DFSE (DFS Embedding). Based on the cut property and the DFSE structure a new algorithm called gdFil (graph duplicate-filtering miner) is introduced.

The outline of this paper is as follows. Section 2 contains some basic concepts and related work. The cut property of the DFS codes, the DFSE structure, and the complete description of gdFil are introduced in section 3. The experimental results are discussed in section 4. Finally, conclusions of the research and some ideas about future directions are exposed in section 5.

2 Background

A *DFS tree* T is constructed when a DFS traversal in a graph $G = \langle V, E\rangle$ is performed. Each DFS traversal (DFS tree) defines a unique order among all the vertices; therefore, we can number each vertex according to this DFS order. Thus, each edge can be represented by a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$ where i and j are the numbers (subindices) of the vertices (v_i and v_j), l_i and l_j are the labels of these vertices respectively, and $l_{(i,j)}$ is the label of the edge connecting v_i and v_j. If $i < j$ it is a forward edge; otherwise it is a backward edge. In short, the order relation $e_1 \prec_e e_2$ holds if e_1 appears before e_2 in a DFS traversal. When e_1 and e_2 have same source and destination vertices, they are compared lexicographically using the order between labels $\prec_l$ (the last three components in each 5-tuple).

The *DFS code* is a sequence of edges built from the DFS tree sorting the edges according to $\prec_e$. The order $\prec_e$ can be also extended to a *lexicographic order* ($\prec_s$) between two DFS codes. The *minimum DFS code* is defined as the minimum sequence among all DFS codes of the same graph according to $\prec_s$ [11].

Suppose that s is a minimum DFS code. An edge e is a *rightmost path extension* of s if e connects the rightmost vertex with another vertex in the rightmost path (backward extension); or it introduces a new vertex connected from a vertex of the rightmost path (forward extension). In such cases, the DFS code $s' = s \diamond e$ is the code obtained extending s by e; s' is called a child of s or s is called a parent of s'.

The set $RE(s)$ of all children of s can be partitioned into several sets $RE(s) = B_0 \cup \ldots \cup B_{n-1} \cup F_0 \cup \ldots \cup F_{n-1}$, where B_i denotes the set that contains the backward extensions to the destination vertex v_i, and F_i is the forward extension set from vertex v_i. For each vertex v_i in the rightmost path, $i \neq n-1$, the edge f_i denotes the forward edge from vertex v_i lying in the rightmost path. The notation e^{-1} is used to refer to the reverse edge of e.

The first two properties introduced in gRed [2] will be named *non-minimality conditions* and the last one *reuse condition*.

1. **Forward non-minimality condition:** If $e \in F_i(s)$ with $i \neq n-1$ and $e \prec_l f_i$, then $s' = s \diamond e$ is a non-minimum DFS code.
2. **Backward non-minimality condition:** If $e \in B_i(s)$ and $e^{-1} \prec_l f_i$, then $s' = s \diamond e$ is a non-minimum DFS code.
3. **Reuse condition:** Let E be one of the sets $F_i(s)$ or $B_i(s)$. If $e, e' \in E$, then, the following statements are true:
   (a) if $s \diamond e$ is a minimum DFS code and $e \preceq_l e'$, then $s \diamond e'$ is a minimum DFS code;
   (b) if $s \diamond e$ is a non-minimum DFS code and $e' \preceq_l e$, then $s \diamond e'$ is a non-minimum DFS code.

The non-minimality conditions were used in gRed to filter the set $RE(s)$ reducing the number of candidates. The reuse condition states that a CF test is not required for some candidates (unlike gSpan where CF tests are performed for each frequent child of s).

3 Frequent connected subgraph mining

In this section, we introduce the cut property of the DFS code which is useful to remove all the duplicate candidates in FCSM before support calculation (unlike gRed where the reuse property are used after the supports of the candidates are calculated [2]). Besides, the DFSE structure will be introduced in the next section. Based on the cut property and the DFSE structure, a new FCSM algorithm called gdFil is presented. The DFSE structure is used to speed up the SI tests getting a better efficiency in the enumeration process and support calculation.

3.1 The cut property of the DFS codes

The cut property is derived from the reuse condition, which is applied to forward and backward extensions as follows:

Theorem 1. *Let s be a minimum DFS code, and E be one of the sets $F_i(s)$ or $B_i(s)$ (the forward or backward extension set for a vertex v_i respectively), and suppose that E is in ascending order according to $\prec_e$. If there are extensions in E that produce non-minimum DFS codes then they are at the beginning of E. Besides, if there are extensions that produce minimum DFS codes they are at the end of E.*

Proof. The proof is omitted due to space restriction.

Thus, each set $F_i(s)$ or $B_i(s)$ is divided in two partitions, the duplicate partition (extensions that produce non-minimum DFS codes) and the useful partition (extensions that produce non-minimum DFS codes). The extensions at duplicate partition are called duplicate extensions and the extensions at useful partition are called useful extensions.

This property states that there could be a non-minimum cut element being the last in the duplicate partition, according to $\prec_e$. Besides, in the same way, there could be a minimum cut element being the first in the useful partition. It should be noticed that these partitions could be empty; therefore, one of those cut elements could not exist. If both elements exist, the non-minimum cut element is the one immediately before the minimum cut element.

3.2 The DFSE structure

Let $D = \{G_0, G_1, \ldots, G_{N-1}\}$ be a graph collection with N graphs which are represented using adjacency lists. Let e be an edge belonging to a DFS code, let $G \in D$ be a graph in the collection, and let $e_k \in E(G)$ be an edge of G where k $(0 \leq k < |E(G)|)$ is the edge identifier. We denote by $e.l_1$, $e.l_e$, and $e.l_2$ the labels of the first vertex, edge and second vertex of e respectively. We say that e and e_k are the same edge according to the labels (denoted as $e =_l e_k$) if $(e.l_1, e.l_e, e.l_2) = (e_k.l_1, e_k.l_e, e_k.l_2)$ or $(e.l_1, e.l_e, e.l_2) = (e_k.l_2, e_k.l_e, e_k.l_1)$. In the first case, we say that $sign(e, e_k) = +1$, and in the second case $sign(e, e_k) = -1$.

If e is a non-symmetric edge $(e.l_1 \neq e.l_2)$, the embedding list (EL) of e regarding G denoted as $L(e, G)$ is defined as follows:

$$L(e, G) = \{(e_k, \epsilon, \lambda) \mid e_k \in E(G), e =_l e_k \text{ and } sign(e, e_k) = \epsilon\} \ , \tag{1}$$

where λ represents a null pointer. The third component in each EL element will be used for those codes with more than one edge. If e is a symmetric edge $(e.l_1 = e.l_2)$, the function $sign(e, e_k)$ can take both values ($+1$ or -1). Thus, we consider in $L(e, G)$ any value $(e_k, \pm 1, \lambda)$ for each $e_k \in E(G)$ such that $e =_l e_k$.

Let s be a minimum DFS code. If s is a code with only one edge e the EL of s regarding G is denoted as $L(s, G) = L(e, G)$.

As it is shown below (2), the EL $L(s \diamond e, G)$ of a child of s is built from $L(s, G)$, during the execution of the procedure `FindEmbedding` (see Fig. 1). The third component in the embedding tuples is a pointer to an embedding tuple of the parent. The notations $\tau.k$, $\tau.\epsilon$ and $\tau.p$ are used to refer to the edge identifier, sign and parent of the tuple τ respectively.

$$\begin{aligned} L(s \diamond e, G) = \{(e_k, \epsilon, p) \mid e_k \in E(G), e =_l e_k, sign(e, e_k) = \epsilon \text{ and } p \text{ is a} \\ \text{pointer to a tuple in } L(s, G)\} \ . \end{aligned} \tag{2}$$

Using the EL definition, we define the DFSE structure of any DFS code s regarding the collection D as:

$$ES(s) = \{(G, L) \mid G \in D, L = L(s, G) \wedge L \neq \emptyset\} \ . \tag{3}$$

```
Procedure MainLoop($D, \delta, S$)
  Input: $D$ - graph collection, $\delta$ - support threshold
  Output: $S$ - mining results

1 Remove infrequent vertices and edges;
2 $S \leftarrow$ all frequent vertices;
3 $S^1 \leftarrow$ all frequent 1-edge codes;
4 forall code $s \in S^1$ do
5     Initialize the DFSE structure $ES(s)$;
6     SubgraphMining($D$,$s$,$\delta$,$S$);
7     $D \leftarrow D \setminus s$;
8     if $|D| < \delta$ then break;
9 end
```

```
Procedure SubgraphMining($D, s, \delta, S$)
  Input: $D$ - graph collection, $s$ - a minimum DFS code,
         $\delta$ - support threshold
  Output: $S$ - mining results

1 $S \leftarrow S \cup \{s\}$;
2 Enumerate($D, s, RE$);
3 forall extension $e \in RE$ do
4     if $s \diamond e$.support $\geq \delta$ then
      SubgraphMining($D, s \diamond e, \delta, S$);
5 end
```

```
Procedure FindEmbedding($s, G, L, REG$)
  Input: $s$ - DFS code, $G$ - a graph of the collection,
  $L$ - the embedding list $L(s, G)$
  Output: $REG$ - the extension set of $s$ in $G$

1  foreach embedding tuple $\tau \in L$ do
2      Mapping $\leftarrow \{(\tau.k, \tau.\epsilon)\}$;
3      $\tau_c \leftarrow \tau$;
4      while $\tau_c.p \neq \lambda$ do
5          $\tau_c \leftarrow \tau_c.p$;
6          Mapping $\leftarrow$ Mapping $\cup \{(\tau_c.k, \tau_c.\epsilon)\}$;
7      end
8      Let $G' \subseteq G$ be the subgraph which is contained in
       Mapping being isomorphic to $s$;
9      forall edge $e_k \in E(G) \setminus E(G')$ such that $e_k$ is
       rightmost extension of $s$ do
10         if $e_k$ is compatible with its computational
           representation in $G$ then $\epsilon \rightarrow +1$;
11         else $\epsilon \rightarrow -1$;
12         $L(s \diamond e_k, G) \leftarrow L(s \diamond e_k, G) \cup \{(e_k, \epsilon, \tau)\}$;
13         $REG \leftarrow REG \cup \{(e_k, L(s \diamond e_k, G))\}$;
14     end
15 end
```

```
Procedure Enumerate($D, s, RE$)
  Input: $D$ - graph collection, $s$ - a minimum DFS code
  Output: $RE$ - the filtered extension set of $s$

1  forall vertex index $i$ in the rightmost path of $s$ do
2      $fMin[i] \leftarrow fNon[i] \leftarrow$ null;
3      $bMin[i] \leftarrow bNon[i] \leftarrow$ null;
4  end
5  forall pair $(G, L) \in ES(s)$ do
6      FindEmbedding($s, G, L, REG$);
7      forall pair $(e, L_e) \in REG$ do
8          if $e$ does not satisfies the non-minimality
           conditions then
9              if ($e$ is a forward edge) then
10                 if IsAllowable($e$, $e.i$, $s$, $fMin$, $fNon$)
                   then
11                     $ES(s \diamond e) \leftarrow ES(s \diamond e) \cup \{(G, L_e)\}$;
12                     $RE \leftarrow RE \cup \{e\}$;
13                 end
14             else
15                 if IsAllowable($e$, $e.j$, $s$, $bMin$, $bNon$)
                   then
16                     $ES(s \diamond e) \leftarrow ES(s \diamond e) \cup \{(G, L_e)\}$;
17                     $RE \leftarrow RE \cup \{e\}$;
18                 end
19             end
20         end
21     end
22 end
```

```
Function IsAllowable($e, i, s, min[\cdot], non[\cdot]$)
  Input: $e$ - an extension, $i$ - a vertex index,
  $min[\cdot]$ and $non[\cdot]$ the cut elements arrays
  Output: true or false

1  if $non[i] =$ null or $e \succ_l non[i]$ then
2      if $min[i] =$ null or $e \prec_l min[i]$ then
3          result $\leftarrow$ isMin ($s \diamond e$);
4          if result then $min[i] = e$;
5          else $non[i] = e$;
6          return result;
7      end
8      else return true;
9  end
10 return false;
```

Fig. 1. Complete description of gdFil.

The use of ES avoids all SI tests since the whole embedding of a candidate can be obtained traversing the third component (a pointer to its parent tuple) of each embedding tuple (see lines 2–8 of FindEmbedding).

3.3 The gdFil Algorithm

The Fig. 1 outlines the pseudo-code of the gdFil algorithm. Note that D represents the graph collection, δ is the support threshold and S contains the mining result. The procedure starts by removing all non-frequent vertices and edges. Next, for each frequent edge its ES is initialized using (3). Later, the procedure `SubgraphMining` is invoked for each frequent edge. At the end of each iteration the edge is dropped from the collection; that is, it will not be used anymore as a possible extension in the next iterations.

The procedure `SubgraphMining` recursively generates all candidate graphs (DFS codes); this process is done while the generated code is frequent. For each minimum DFS code s, its extension set RE is calculated using the procedure `Enumerate`. In gdFil the set RE only contains useful extensions, since duplicate extensions are eliminated in `Enumerate`. This is the a difference of gdFil with regard to gSpan and gRed.

In the procedure `Enumerate`, all occurrences of each DFS code s in the graph collection are scanned to detect only extensions that occur in the collection (see lines 5–7). The structure $ES(s)$ is scanned using `FindEmbedding` to efficiently locate the possible children. The lines 11 and 16 create and update the child ES.

The non-minimality conditions were also used in gdFil to filter directly some duplicated extensions (see line 8 of `Enumerate`). The other main contribution of gdFil lies in the lines 9–19 since the function `IsAllowable` allows to filter all duplicate candidates. During the scanning to keep the current cut elements, four arrays $fMin$, $fNon$, $bMin$, and $bNon$ are used. The elements in these arrays represent the non-minimum and minimum cuts for forward and backward extension sets. These elements are initialized as null (see lines 1–4 of `Enumerate`). Each vertex v_i in the rightmost path of s is represented in one position of each array.

The function `IsAllowable` uses the cut property to decide if several extensions are useful or duplicate, without CF test (see lines 1–2). Nevertheless, if an element is, according to $\prec_e$, after than the non-minimum cut and before than the minimum cut it should be tested through `isMin`(s) function, to decide if it represents a minimum DFS code. In this case, the cut elements are updated according to the test result. The function `isMin`(s) is the same function used in gSpan and gRed [11, 2]. The lines 10 and 15 of `Enumerate`, invoke the function `IsAllowable` for the forward and backward extensions respectively. The fourth and fifth arguments must be chosen according to the case. The second argument is the initial vertex for forward extensions or the destination vertex for backward extensions, see the definition of $F_i(s)$ and $B_i(s)$ in section 2.

The DFSE structures are calculated and stored only for useful candidates since the enumeration strategy of gdFil allows to remove all duplicate candidates.

4 Experimental results

We compared gdFil against gRed, gSpan and Gaston; the last two algorithms were taken from the Java framework [10] which is distributed under GNU license, gRed and gdFil were implemented by us and their implementations are compatible with this framework. All the experiments were done using an Intel Core 2 Duo PC at 2.2 GHz with 4 GB of RAM using a 64-bits Debian GNU/Linux distribution. The IBM Java Virtual Machine (JVM) Version 6 was used to run the algorithms. The maximum heap memory space of JVM was assigned in 3.2GB. Thus, we remove the influence of operative system swap operations during the executions.

The CAN2DA99[3], HIV[4], and the entire NCI[5] datasets were used to determine how the algorithms scale. These graph collections have been commonly used in different works for performance evaluations [10].

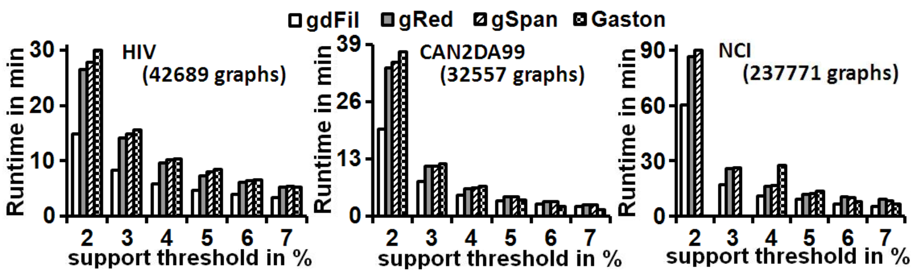

Fig. 2. Runtime with CAN2DA99, HIV, and NCI datasets varying the support threshold.

The algorithms were compared regarding their runtimes on the three collections varying the support threshold (see Fig. 2). In order to illustrate how the algorithms scale with a lot of candidates, only low support thresholds were considered. These kind of thresholds are very important in data mining applications. For example, there are some applications like classification and clustering where frequent complex graph structure are important [4], and these complex structures only can be found with low support thresholds.

The runtime rises for Gaston for the lowest support thresholds and it was unable to complete the execution for support thresholds less than 4% in NCI due to memory requirements. Gaston needed much more memory than the other tested algorithms, since it uses ES for useful and duplicated candidates. The best runtimes were obtained by our algorithm because it does not store any duplicate candidate and achieve fast isomorphism tests using the DFSE structure. It is known that much of the time consumption in gSpan and gRed is used by SI tests during the candidate enumeration process. Since gdFil can achieve fast SI tests using the DFSE structure it got the best runtimes.

5 Conclusions

In this paper, a cut property of the DFS code which is useful to remove all the duplicate candidates in FCSM during the candidate enumeration was intro-

[3] http://dtp.nci.nih.gov/docs/cancer/cancer_data.html
[4] http://dtp.nci.nih.gov/docs/aids/aids_data.html
[5] http://cactus.nci.nih.gov/ncidb2/download.html

duced. This property allows to define boundaries between useful and duplicate candidates during the pattern growth process.

A new FCSM algorithm called gdFil was designed using the cut property. Besides, we introduce a new ES, called DFSE, to reduce the cost of SI tests.

We compared gdFil against gRed, gSpan and Gaston. The experimentation showed that our proposal achieves the best performance.

As future work, we are going to develop new ways for taking advantage of the cut property and the DFSE structure in order to achieve better performance in graph mining.

References

1. Borgelt, C., Berthold, M.R.: Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In: Proceedings of the International Conference on Data Mining (ICDM'02), Maebashi, Japan, pp. 211–218, 2002.
2. Gago, A., Medina, J.E., Carrasco, J.A., Martínez, J.F.: Mining Frequent Subgrahps Reducing the Number of Candidates. In: Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD'08), Antwerp, Belgium, pp. 365–376, 2008.
3. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent Pattern Mining: Current Status and Future Directions. Data Mining and Knowledge Discovery, 10th Anniversary Issue, Vol. 15 (1): pp. 55–86, 2007.
4. Hossain, M., Angryk, R.: GDClust: A Graph-based Document Clustering Technique. In *Proceedings of the 7th IEEE International Conference on Data Mining Workshops*, Omaha, NE, pp. 417–422, 2007.
5. Huan, J., Wang, W., Prins, J.: Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. In: Proceedings of the International Conference on Data Mining (ICDM'03), Melbourne, FL, pp. 549–552, 2003.
6. Inokuchi, A., Washio, T., Motoda, H.: An Apriori based Algorithm for Mining Frequent Substructures from Graph Data. In: Proceedings of the European Symposium on the Principle of Data Mining and Knowledge Discovery (PKDD'00), Lyon, France, pp. 13–23, 2000.
7. Inokuchi, A., Washio, T., Nishimura, K., Motoda, H.: A Fast Algorithm for Mining Frequent Connected Subgraphs. Technical Report RT0448, IBM Research, Tokyo Research Laboratory, 2002.
8. Kuramochi, M., Karypis, G.: Frequent Subgraph Discovery. In: Proceedings of the International Conference on Data Mining (ICDM'01), San Jose, CA, pp. 313–320, 2001.
9. Nijssen, S., Kok, J.: A Quickstart in Frequent Structure Mining can Make a Difference. In: Proceedings of the ACM SIGKDD International Conference on Kowledge Discovery in Databases (KDD'04), Seattle, WA, pp. 647–352, 2004.
10. Wörlein, M., Meinl, T., Fischer, I., Philippsen, M.: A Quantitative Comparison of the Subgraph Miners Mofa, gSpan, FFSM, and Gaston. In: Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'05), Porto, Portugal, pp. 392–403, 2005.
11. Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In: Proceedings of the International Conference on Data Mining (ICDM'02), Maebashi, Japan, pp. 721–724, 2002.

Comments describing the changes

1. The name of section 2 was changed to “Background” attending the suggestion 1 and 3 of the second reviewer.
2. The size of Fig. 2 was increased in order to improve the image quality attending the suggestion 2 of the second reviewer.
3. The last paragraph of section 3 was summarized in order to free space. It allows to include a new paragraph before the second last paragraph of the introduction. This new paragraph contains the motivation (suggestion 4 of the second reviewer) of our work since we do not have more space to include a complete motivation section.
4. The discussion about experimental results (the last suggestion of the second reviewer) was not expanded due to lack of space.