Mining Frequent Connected Subgraphs Reducing the Number of Candidates

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

[1] Data Mining Departament,
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. In this paper, a new algorithm for mining frequent connected subgraphs called gRed (graph Candidate Reduction Miner) is presented. This algorithm is based on the gSpan algorithm proposed by Yan and Jan. In this method, the mining process is optimized introducing new heuristics to reduce the number of candidates. The performance of gRed is compared against two of the most popular and efficient algorithms available in the literature (gSpan and Gaston). The experimentation on real world databases shows the performance of our proposal overcoming gSpan, and achieving better performance than Gaston for low minimal support when databases are large.

1 Introduction

Nowadays, due to the rapid scientific and technological advances, there are great creation, storage and data distribution capacities. This situation has boosted the necessity of new tools to transform this big amount of data into useful information or knowledge for decision makers. When these data are complex and structured, this transformation requires techniques that usually have high time and memory requirements. Examples of these techniques are those related to frequent subgraph mining; i.e., the process of finding subgraphs that occur frequently in a collection of graphs.

Frequent subgraph mining has become an important topic in data mining researches with wide applications [3], including mining substructures from chemical compound databases, XML documents, citation networks, biological networks, etc. As consequence several algorithms have been proposed to find all frequent connected subgraphs in collections of labeled graphs [6,7,11,2,4,8].

Labeled graphs can be used to model relations among data in the aforementioned applications because labels can represent attributes of entities and relations among themselves. For example in chemistry, the different kinds of atoms and bonds in a chemical compound can be modeled by vertex and edge labels respectively.

The first frequent subgraph miner called AGM was introduced by Inokuchi *et al.* for unconnected graphs [5]. This algorithm was followed by the FSG algorithm [7] and AcGM [6] (an adaptation of AGM), for mining frequent connected subgraphs. These algorithms have the same setup as the original Apriori algorithm for mining frequent itemsets [1].

To avoid supposed overheads incurred in the earlier algorithms, new pattern growth based algorithms such as gSpan [11], MoFa [2], FFSM [4] and Gaston [8] were developed. In [10] these algorithms were compared in a common framework. In this experimentation, the four algorithms were competitive among themselves, although Gaston and MoFa were the fastest and slowest algorithms respectively, in almost all tests. On the other hand, gSpan was the best algorithm regarding its memory requirements. The embedding structures used by MoFa, FFSM and Gaston could cause problems if not enough memory is available or if the memory throughput is not high enough.

In this paper, a new pattern growth algorithm called gRed (graph Candidate Reduction Miner) is introduced. This algorithm is based on the gSpan scheme; but using novel properties of the DFS code that allows to reduce the number of candidates for optimizing the mining process.

The basic outline of this paper is as follows. Section 2 is dedicated to the related work, it includes the basic concepts introduced by gSpan algorithm. The details of gRed algorithm are discussed in the section 3. The experimental results are presented in section 4. Conclusions of the research and some ideas about future directions are exposed in section 5.

2 Preliminaries

In gRed, each candidate graph is represented by its minimum DFS (Depth First Search) code. This kind of canonical representation, based on DFS graph traversal, was introduced in gSpan [11]. Some concepts introduced in gSpan are required for understanding our algorithm; therefore, we include them in section 2.1.

2.1 Basic gSpan Concepts

The basic concepts we will use in this paper are the following.

DFS Tree. A DFS tree T is constructed when a DFS traversal in a graph $G = \langle V, E \rangle$ is performed. G can have different DFS trees because there are more than one DFS traversal. Each DFS traversal (DFS tree) defines a unique order among all the vertices; therefore, we can number each vertex according to this DFS order. The root and the right most vertex in T are v_0 and v_n respectively. The right most path is the straight path from v_0 to v_n in T. The forward edge set $F(T)$ contains all the edges in T, and the backward edge set $B(T)$ contains the edges which are not in T.

```
Procedure MainLoop($D$, $\delta$, $S$)
Input: $D$ - database, $\delta$ - support threshold
Output: $S$ - mining results
1 Remove infrequent vertices and edges;
2 $S \leftarrow S^1 \leftarrow$ all frequent 1-edge codes
3 forall code $s \in S^1$ do
4     Initialize the TID list $s.L$ by the graphs which contains the edge of $s$;
5     SubgraphMining($D$,$s$,$\delta$,$S$);
6     $D \leftarrow D \setminus e$;
7     if $|D| < \delta$ then break;
8 end
```

```
Procedure SubgraphMining($D$, $s$, $\delta$, $S$)
1 if not isMin($s$) then return;
2 $S \leftarrow S \cup \{s\}$;
3 Enumerate($D$, $s$, $RE$);
4 forall edge extension $e \in RE$ do
5     if $s \diamond e$.support $\geq \delta$ then SubgraphMining($D$, $s \diamond e$, $\delta$, $S$);
6 end
```

```
Procedure Enumerate($D$, $s$, $RE$)
1 forall graph $g \in s.L$ do
2     Enumerate the next occurrence of $s$ in $g$;
3     forall right most extension $e$ of $s$ do
4         $s \diamond e.L \leftarrow s \diamond e.L \cup \{g\}$;
5         $RE \leftarrow RE \cup \{e\}$;
6     end
7     if any occurrence of $s$ in $g$ are not covered then goto line 2;
8 end
```

Fig. 1. General description of gSpan algorithm

DFS Code. The DFS code is a sequence of edges built from the DFS Tree. This sequence is obtained considering the destination vertices in $F(T)$ according to the DFS order. The backward edges from a vertex are inserted just before its forward edges; if the vertex has several backward edges, these are included in the DFS order of their destination vertices. Multiple edges between same vertices are ordered according to the lexicographic order ($\prec_l$) of its labels. These considerations for building the sequence define the following linear order ($\prec_e$) between two edges.

DFS lexicographic order. For simplicity, each edge can be represented by a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$ where i and j are the 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. If $i < j$ it is a forward edge; otherwise it is a backward edge.

In summary, the inequality $e_1 \prec_e e_2$ holds (assume that $e_1 = (i_1, j_1, \ldots)$ and $e_2 = (i_2, j_2, \ldots)$) if and only if one of the following statements is true:

- $e_1, e_2 \in F(T)$ and $j_1 < j_2$ or $i_1 > i_2 \wedge j_1 = j_2$;
- $e_1, e_2 \in B(T)$ and $i_1 < i_2$ or $i_1 = i_2 \wedge j_1 < j_2$;
- $e_1 \in B(T)$, $e_2 \in F(T)$ and $i_1 < j_2$;
- $e_1 \in F(T)$, $e_2 \in B(T)$ and $j_1 \leq i_2$;
- $i_1 = i_2$, $j_1 = j_2$ and $e_1 \prec_l e_2$.

The lexicographic order $\prec_l$ compares the edges e_1 and e_2 regarding the last three components in each 5-tuple. The vertex label l_i gets first priority, the edge label $l_{(i,j)}$ gets the second, and the vertex label l_j gets the third to determine the order between two edges.

The order $\prec_e$ can be also extended to a lexicographic order ($\prec_s$) to compare two edge sequences (two DFS codes). Let $s_1 = (a_1, a_2, \ldots, a_m)$ and $s_2 = (b_1, b_2, \ldots, b_n)$ be two DFS codes. We say that $s_1 \prec_s s_2$ if one of the following conditions is true:

$$\exists t, \forall k < t, a_k = b_k, \text{ and } a_t \prec_e b_t \; ; \tag{1}$$

$$m < n \text{ and } \forall k \leq m, a_k = b_k \; . \tag{2}$$

Minimum DFS Code. It is defined as the minimum sequence according to the order $\prec_s$ among all DFS codes of the same graph.

Rightmost path extension. Given a DFS code s and an edge e, 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'.

gSpan guarantees the completeness of mining results only working with the minimum DFS codes, pruning non minimal children in the solution space. Fig. 1 describes the pseudo-code of gSpan. This pseudo-code is an integration of the algorithm descriptions presented in [11,12].

All pattern growth algorithms generate duplicated candidates during the enumeration process. In gSpan, the duplicated candidates are non-minimal codes. Instead of calculating the minimum DFS code of s from all possible DFS codes, picking up the smallest one and comparing it against s, gSpan defines a more efficient function `isMin`(s) in line 1 of `SubgraphMining`. A heuristic search was designed using the DFS lexicographic order. Whenever some prefix of a DFS is generated and it is less than s, then s is not minimal and the search concludes.

For support calculation and candidate enumeration, gSpan uses a TID list. The TID list (Transaction ID list) contains the ID of each graph in the database that holds the corresponding subgraph. In the procedure `Enumerate`, $s.L$ is used to determine the possible extension set for s, performing subgraph isomorphism tests to find all the embeddings of s in each graph in $s.L$. In line 5 of `SubgraphMining`, the support of $s \diamond e$ is the length of $s \diamond e.L$.

3 The gRed Algorithm

The gRed algorithm can also be described using the pseudo-code of Fig. 1. Only the procedures `SubgraphMining` and `Enumerate` are changed by new procedures `gRed-SubgraphMining` and `gRed-Enumerate` respectively. Novel properties of the DFS codes introduced in the following section are used by the new procedures for optimizing the mining process.

3.1 DFS Codes

Suppose that $s = e_0, e_1, \ldots, e_m$ is a minimum DFS code. The set $RE(s)$ of all rightmost path extensions (see section 2.1) of s can be partitioned into three sets $B(s)$ (backward extensions), $FV(s)$ (forward extensions from the rightmost vertex) and $FN(s)$ (forward extensions from a non rightmost vertex in the right most path), $RE(s) = B(s) \cup FV(s) \cup FN(s)$ (see Fig. 2). Thus, the forward extensions can be represented by $F(s) = FV(s) \cup FN(s)$.

Let v_i be a vertex in the right most path, let v_n be the right most vertex in s and let $F_i(s)$ be the forward extension set from vertex v_i. If $v_i \neq v_n$ we can use $FN_i(s)$ to refer to $F_i(s)$. Similarly, we use $B_i(s)$ to denote the set that contains the backward extensions to the destination vertex v_i. For example, in Fig. 2 the right most path is (v_0, v_4, v_6); therefore, we have $FV(s) = F_6(s)$, $FN(s) = F_0(s) \cup F_4(s)$ and $B(s) = B_0(s) \cup B_4(s)$.

If $v_i \neq v_n$, we denote f_i as the forward edge from v_i lying in the right most path. For each edge e, we use e^{-1} to refer to the reverse edge of e. For example in Fig. 2, we have $f_4 = (4, 6, B, \_, C)$ and $f_4^{-1} = (6, 4, C, \_, B)$.

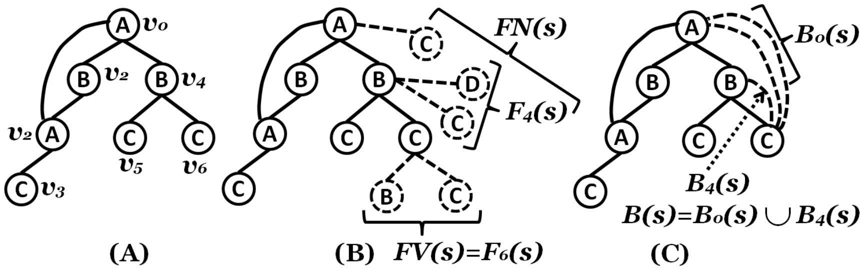

Fig. 2. Partitions in the rightmost path extensions, (A) example of a DFS tree, (B) the forward extensions $FV(s) \cup FN(s)$ and (C) the backward extensions $B(s)$

The following two results are sufficient conditions for a child to be non minimal. Both statements can be used by the procedure `gRed-Enumerate` to filter the possible extensions from a DFS code s.

Proposition 1. *Let s be a minimal DFS code. If $e \in FN_i(s)$ and $e \prec_l f_i$, then $s' = s \diamond e$ is a non-minimal child of s.*

Proof. Let h be an integer number such that $e_h = f_i$. The edges e_h and e start from v_i and they are forward edges in s'. Therefore, we can perform a DFS traversal visiting first e and then e_h or vice versa. If e is visited immediately before than e_h, the resulting DFS code has the following format $s'_1 = e_0, \ldots, e_{h-1}, e, e'_h, \ldots, e'_m$, where e'_j is e_j with another subindices for each $j \geq h$. The codes s' and s'_1 have the same prefix $e_0, \ldots, e_{h-1}$ and we are considering that $e \prec_l e_h$; therefore, by the condition (1), $s'_1 \prec_s s'$. Thus, we conclude that s' is a non-minimal child of s. □

The proposition 1 allows to characterize the non-minimality of certain forward extensions. Similar results for backward extensions are showed in proposition 2.

Proposition 2. *Let s be a minimal DFS code. If $e \in B_i(s)$ and $e^{-1} \prec_l f_i$, then $s' = s \diamond e$ is a non-minimal child of s.*

Proof. Similarly to the proof of the proposition 1, h is the integer number such that $e_h = f_i$. We can perform a DFS traversal visiting first e^{-1} and then e_h or vice versa. If e^{-1} is visited immediately before than e_h, the resulting DFS code have the following format $s'_1 = e_0, \ldots, e_{h-1}, e^{-1}, e'_h, \ldots, e'_m$. The codes s' and s'_1 have the same prefix $e_0, \ldots, e_{h-1}$ and we assume that $e^{-1} \prec_l e_h$; therefore, $s'_1 \prec_s s'$. Thus, we conclude that s' is a non-minimal child of s. □

In gSpan, a duplicate test (minimality test) is performed for each frequent child of s, i.e. the number of such tests is $|RE(s)|$. Let $RE_0(s)$ be the extension set obtained from $RE(s)$ by removing the extensions whose non-minimality is guaranteed according to the propositions 1 and 2. The algorithm gRed only considers the extensions in $RE_0(s)$.

It is not always necessary to perform the duplicate test for each child of s in $RE_0(s)$. The following propositions allow to avoid some minimality tests in the context of the DFS codes. The procedure `gRed-SubgraphMining` uses this properties to speedup the mining process.

Proposition 3. *Let s be a minimal DFS code and let $e, e' \in F_i(s)$ be two forward extensions of s by the same vertex v_i. Then, the following statements are true:*

1. *if $s \diamond e$ is a minimal child and $e \preceq_l e'$, then $s \diamond e'$ is a minimal child;*
2. *if $s \diamond e$ is a non-minimal child and $e' \preceq_l e$, then $s \diamond e'$ is a non-minimal child.*

Proof. Let us proof separately each case. We assume that s is the edge sequence $e_0, e_1, \ldots, e_m$.

In the first case, we have that $s \diamond e$ is a minimal child and $e \preceq_s e'$. Suppose that $s \diamond e'$ is a non-minimal child, then there is at least one code $s_1 = a_0, a_1, \ldots, a_{m+1}$ such that $s_1 \prec_s s \diamond e'$. Using the definition of $\prec_s$ (see section 2.1), there is an integer t, $0 \leq t \leq m$ such that $a_k = e_k$ for all $k < t$, and $a_t \prec_e e_t$. As it can be noticed, $t < m + 1$ because s is a minimal DFS code. Thus, by the condition (1), $s_1 \prec_s s$.

Since e and e' start from the same vertex, we can replace the edge representing e' in s_1 by the edge e. Assume s_1 as the code obtained when replacing of e' by e in

s_1; this code is a valid one for the graph coded by $s \diamond e$ and we have $s_1' \prec_s s_1 \prec_s s$. Using the condition (2), $s_1' \prec_s s \prec_s s \diamond e$. Then, $s \diamond e$ is a non-minimal child of s, representing a contradiction. Therefore, the initial assumption ($s \diamond e'$ is a non-minimal child) must be false. Thus, we conclude the proof for the first case.

In the second case, we have that $s \diamond e$ is a non-minimal child and $e' \preceq_s e$. Then, there is at least one code $s_1 = a_0, a_1, \ldots, a_{m+1}$ such that $s_1 \prec_s s \diamond e'$. Let t be the integer such that $0 \leq t \leq m$, $a_k = e_k$ for all $k < t$, and $a_t \prec_e e_t$. Thus, by the condition (1), we have $s_1 \prec_s s$. Since e and e' start from the same vertex, we can replace the edge representing e in s_1 by the edge e'. The resulting code (assume it is s_1') is a valid DFS code for the graph coded by $s \diamond e'$ and we have $s_1' \prec_s s_1 \prec_s s \prec_s s \diamond e'$. Therefore, $s \diamond e'$ is a non-minimal child. □

Proposition 4. *Let s be a minimal DFS code and let $e, e' \in B_i(s)$ be two backward extensions of s with destination vertex v_i. Then, the following statements are true:*

1. *if $s \diamond e$ is a minimal child and $e \preceq_l e'$, then $s \diamond e'$ is a minimal child;*
2. *if $s \diamond e$ is a non-minimal child and $e' \preceq_l e$, then $s \diamond e'$ is a non-minimal child.*

Proof. The proof is similar to that given for proposition 3. □

The last two propositions state that the minimality tests are not required in some elements of $B_i(s)$ or $F_i(s)$. The extensions in $RE_0(s)$ are sorted according to the aforementioned order $\prec_e$, firstly the elements in $B_0(s), \ldots, B_n(s)$, and finally the elements in $F_n(s), \ldots, F_0(s)$. Each set $B_i(s)$ or $F_i(s)$ is ordered internally by the lexicographic order $\prec_l$. Under the proposition 3, we only need to find the minimal extension $e \in F_i(s)$ such that its corresponding predecessor extensions in $F_i(s)$ are non-minimal. The successors of e in $F_i(s)$ are also minimal extensions; therefore, the minimality tests are not required. Similar observations might be indicated for each $B_i(s)$ using the proposition 4.

These four propositions were not used in the original gSpan algorithm described in [11,12]. In the following section we illustrate how the aforementioned properties are used to design the gRed algorithm.

3.2 The Algorithm

The Fig. 3 outlines the pseudo-code of the gRed algorithm. Note that D represents the graph database, δ is the minimum support threshold and S contains the mining result.

`gRed-MainLoop` is quite similar to the procedure `MainLoop` of gSpan. It starts by removing all infrequent vertices and edges. Next, for each frequent edge its TID list is initialized before the `gRed-SubgraphMining` is invoked. At the end of each iteration the edge is dropped from the database, i.e. it will not be used as possible extensions in the next iterations.

The procedure `gRed-SubgraphMining` recursively generates all candidate codes (graphs), this process is done while the generated code is frequent. Firstly, for each minimum DFS code s its extension set $ER_0(s)$ is calculated using the procedure `gRed-Enumerate`. The propositions 3 and 4 are used in `gRed SubgraphMining`

```
Procedure gRed-MainLoop($D$, $\delta$, $S$)
Input: $D$ - database, $\delta$ - support threshold
Output: $S$ - mining results
1  Remove infrequent vertices and edges;
2  $S \leftarrow S^1 \leftarrow$ all frequent 1-edge codes
3  forall code $s \in S^1$ do
4      Initialize the TID list $s.L$ by the graphs which contains the edge of $s$;
5      gRed-SubgraphMining($D$,$s$,$\delta$,$S$);
6      $D \leftarrow D \setminus e$;
7      if $|D| < \delta$ then break;
8  end
```

```
Procedure gRed-SubgraphMining($D$, $s$, $\delta$, $S$)
1  $S \leftarrow S \cup \{s\}$;
2  gRed-Enumerate($D$, $s$, $RE$);
3  forall extensions set $E$, might be $RE.B_i$ or $RE.F_i$ do
4      Scan the first elements in $E$ according to the order $\prec_e$, removing the
       non-minimal extensions;
5      forall extension $e \in E$ do
6          if $s \diamond e$.support $\geq \delta$ then gRed-SubgraphMining($D$, $s \diamond e$, $\delta$, $S$);
7      end
8  end
```

```
Procedure gRed-Enumerate($D$, $s$, $RE$)
1  forall graph $g \in s.L$ do
2      Enumerate the next occurrence of $s$ in $g$;
3      forall right most extension $e$ of $s$ do
4          $i \leftarrow e.i$;
5          $f_i \leftarrow$ the forward edge starting in $i$ and lies in the right most path;
6          if $e$ is a forward edge and $e \geq_l f_i$ then        /* See proposition 1 */
7              $s \diamond e.L \leftarrow s \diamond e.L \cup \{g\}$;
8              $RE.F_i \leftarrow RE.F_i \cup \{e\}$;
9          end
10         if $e$ is a backward edge and $e^{-1} \geq_l f_i$ then   /* See proposition 2 */
11             $s \diamond e.L \leftarrow s \diamond e.L \cup \{g\}$;
12             $RE.B_i \leftarrow RE.B_i \cup \{e\}$;
13         end
14     end
15     if any occurrence of $s$ in $g$ are not covered then goto line 2;
16 end
```

Fig. 3. General description of gRed algorithm

for reducing the number of expensive minimality tests regarding gSpan and guaranteeing the completeness in the mining result. The minimality test in line 4 is performed using the same isMin(s) function used in gSpan (see section 2.1). In gRed the `isMin`(s) function is used only for the first extensions of each $B_i(s)$ or $F_i(s)$ while in gSpan the test is performed for every child of s.

In the procedure `gRed-Enumerate`, all occurrences of s in each graph of the TID list $s.L$ are enumerated. Thus, all the possible extensions of s in the database are generated. Using the propositions 1 and 2, some non-minimum extensions are filtered. For the specific sub-graph isomorphism testing procedure of line 2, we use the same enumerating engine proposed by gSpan in [12].

4 Experimental Results

All the experiments were done using an Intel Core 2 Duo PC at 2.2 GHz with 2 GB of RAM. We compare gRed only against gSpan and Gaston; both were implemented in a common Java framework [10] which is distributed under GNU license. Our implementation of gRed is compatible with this framework. The SUN Java Virtual Machine (JVM) 1.5.0 was used to run the algorithms.

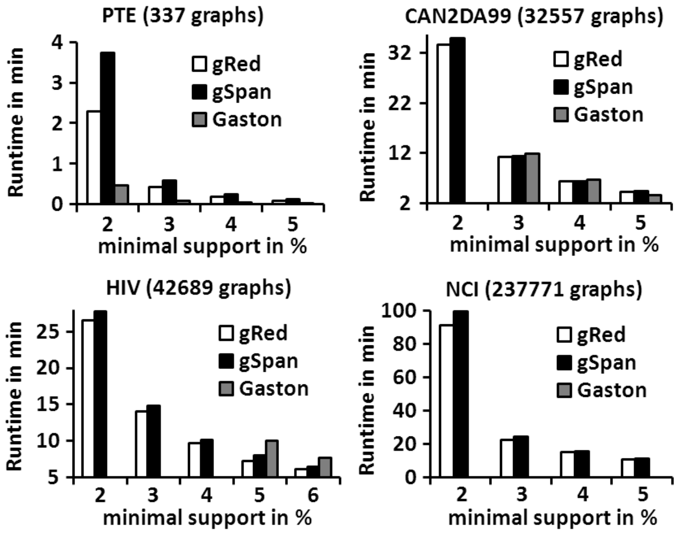

Fig. 4. Runtime with datasets PTE, CAN2DA99, HIV and NCI varying the minimum support

First of all, in order to show that the algorithm implementations get approximately the same results as those published in [8,10] we retested the algorithms as in those works. The performance was retested using PTE [9] and CAN2DA99[1] datasets. The results in both datasets are included adding the performance of gRed.

[1] http://dtp.nci.nih.gov/docs/cancer/cancer_data.html

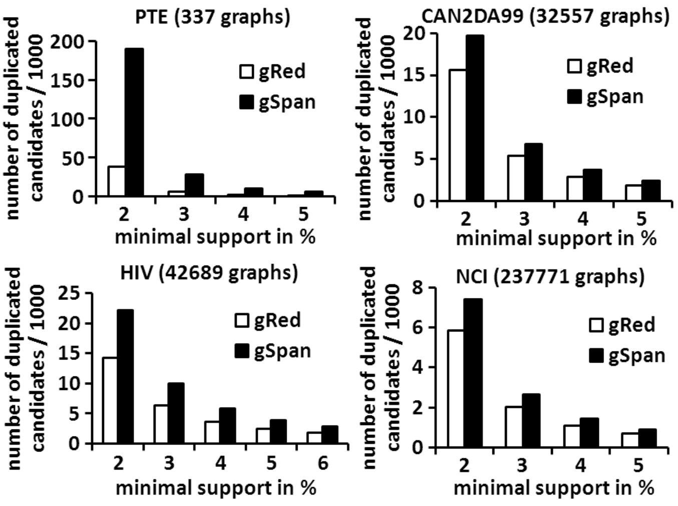

Fig. 5. The number of duplicated candidates found in datasets PTE, CAN2DA99, HIV and NCI varying the minimum support

The NCI[2] and HIV[3] datasets are used to determine how the algorithms scale when the database size increases. Commonly, these four datasets have been used in different works for performance evaluations [10].

The runtime for the algorithms was recorded varying the support threshold for the four datasets. A comparison of gRed, gSpan and Gaston regarding its execution times are showed in Fig. 4. In order to illustrate how the algorithms scale with a lot of candidates, only low support thresholds were considered. The runtime rises for Gaston on large databases (CAN2DA99, HIV and NCI) for these minimal supports. Besides, Gaston was unable to complete the execution for low minimal supports (less than 3% in CAN2DA99, 5% in HIV and 6% in NCI) due to memory requirements. Gaston needed much more memory than the other tested algorithms (see Fig. 6). However, in the smallest database (PTE), the best results were achieved by Gaston. The best runtimes on the large databases were obtained by gRed and gSpan for the evaluated support thresholds.

As we can see, gRed beats gSpan in all tests. It is known that much of the time consumption in gSpan is used by subgraph isomorphism tests during the candidate enumeration process. Since gRed also uses this kind of tests, it has similar behavior. However, gRed shows significant improvements when the database is large. In PTE the improvement achieved by gRed is even greater since the number of duplicate candidates declines considerably (see Fig. 5).

[2] http://cactus.nci.nih.gov/ncidb2/download.html
[3] http://dtp.nci.nih.gov/docs/aids/aids_data.html

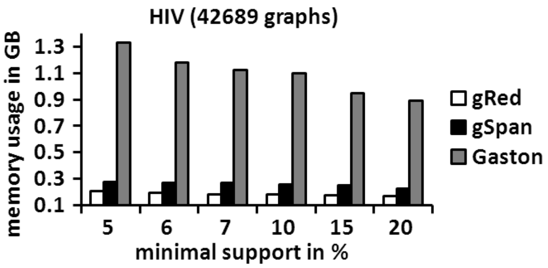

Fig. 6. Memory usage on the HIV dataset varying the minimum support

The algorithms gRed and gSpan are also compared regarding the number of duplicated candidates in Fig. 5. The pruning strategies used to minimize the number of duplicates in Gaston is very different from those used by gRed and gSpan. Gaston is not included in this comparison to highlight the differences between gRed and gSpan. The number of duplicates in all cases were significantly reduced by gRed; even in PTE for minimal support = 2, it reduces almost 60% of the duplicates regarding gSpan. This improvement corresponds to the runtimes of Fig. 4; nevertheless, the runtime improvement was not even greater because subgraph isomorphism tests are time-consuming. This result suggests that, if gRed is combined with the evaluation strategies of the other algorithms (for example the embedding structures used in Gaston), we might achieve better runtime scores.

The memory consumption was recorded varying the support threshold on the HIV dataset (see Fig. 6). We choose HIV in order to show an example of the memory problem of Gaston for low minimal support in correspondence to the runtimes of Fig. 4. The improvement of gRed regarding memory requirement can be appreciated in Fig. 6.

5 Conclusions

In this paper, a new algorithm called gRed, for frequent connected subgraph mining, was introduced. Novel properties of the DFS code, that allow to reduce the number of candidates during the mining process, were studied and implemented. In this research, we show that the DFS code has not been sufficiently studied and new properties can be found to improve the mining process.

We compared gRed against two reported algorithms. The experimentation showed that our proposal overcome gSpan in every tests. In the experiments, gRed and gSpan achieved better performance evaluations than Gaston for low minimal support when databases are large.

As future work, we are going to develop hybrid approaches of gRed in combination with evaluation strategies of other algorithms like Gaston.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proceedings of the 1994 International Conference on Very Large Data Bases (VLDB 1994), Santiago, Chile, pp. 487–499 (1994)
2. Borgelt, C., Berthold, M.R.: Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In: Proceedings of the 2002 International Conference on Data Mining (ICDM 2002), Maebashi, Japan, pp. 211–218 (2002)
3. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent Pattern Mining: Current Status and Future Directions. In: Data Mining and Knowledge Discovery (DMKD 2007), 10th Anniversary Issue, vol. 15(1), pp. 55–86 (2007)
4. Huan, J., Wang, W., Prins, J.: Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. In: Proceedings of the 2003 International Conference on Data Mining (ICDM 2003), Melbourne, FL, pp. 549–552 (2003)
5. Inokuchi, A., Washio, T., Motoda, H.: An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 13–23. Springer, Heidelberg (2000)
6. Inokuchi, A., Washio, T.: Nishimura and K., Motoda, H.: A Fast Algorithm for Mining Frequent Connected Subgraphs, Technical Report RT0448. In IBM Research, Tokyo Research Laboratory, pp. 10 (2002)
7. Kuramochi, M., Karypis, G.: Frequent Subgraph Discovery. In: Proceedings of the 2001 International Conference on Data Mining (ICDM 2001), San Jose, CA, pp. 313–320 (2001)
8. Nijssen, S., Kok, J.: A Quickstart in Frequent Structure Mining can Make a Difference. In: Proceedings of the 2004 ACM SIGKDD International Conference on Kowledge Discovery in Databases (KDD 2004), Seattle, WA, pp. 647–352 (2004)
9. Srinivasan, A., King, R.D., Muggleton, S.H., Sternberg, M.: The Predictive Toxicologic Evaluation Challenge. In: Proceedings of the 15th International Conference on Artificial Intelligence (IJCAI 1997), pp. 1–6. Morgan-Kaufmann, San Francisco (1997)
10. Wörlein, M., Meinl, T., Fischer, I., Philippsen, M.: A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. In: Jorge, A.M., Torgo, L., Brazdil, P.B., Camacho, R., Gama, J. (eds.) PKDD 2005. LNCS (LNAI), vol. 3721, pp. 392–403. Springer, Heidelberg (2005)
11. Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In: Proceedings of the 2002 International Conference on Data Mining (ICDM 2002), Maebashi, Japan, pp. 721–724 (2002)
12. Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining, Expanded Version, UIUC Technical Report, UIUCDCS-R-2002-2296 (2002)