Subgraph Isomorphism Detection using a Code Based Representation

Ivan Omos[1], Jesus A. Gonzalez[1] and Mauricio Osorio[2]

[1]Instituto Nacional de Astrofísica, Óptica y Electrónica
Luis Enrique Erro #1, Sta. Maria Tonanzintla, Puebla, México
{iolmos, jagonzalez}@inaoep.mx
[2]Universidad de las Américas Puebla
Sta. Catarina Mártir, Puebla, México
josorio@mail.udlap.mx

Abstract

Subgraph Isomorphism Detection is an important problem for several computer science subfields, where a graph-based representation is used. In this research we present a new approach to find a Subgraph Isomorphism (SI) using a list code based representation without candidate generation. We implement a step by step expansion model with a width-depth search. Our experiments show a promising method to be used with scalable graph matching tools to find interesting patterns in Machine Learning and Data Mining applications.

Introduction

The problem to find similarities between graphs is an important problem in several areas such as Machine Learning and Data Mining, where repetitive patterns are used to classify or summarize information. This problem is also known as graph matching, which includes tasks as graph / subgraph isomorphism and frequent subgraph detection (Cook and Holder 1994, Xifeng and Jiawei 2002, Shayan 2002).

The core of the problem consists on finding the associations or mappings between the vertices and edges of the graphs to match. However the task is not easy, because in general the number of possible mappings increases exponentially with respect to the graph dimension and density (#vertices and #edges). Moreover, it is known that the Subgraph Isomorphism Problem (SIP) is NP – complete and then, in the worst case the time to detect a mapping that represents the isomorphism is exponential, unless P = NP (Garey and Johnson 2003). However, if a problem is NP – complete, means that only some instances fall in the worst case, but not necessary all of them.

Several works have been developed to find the mappings, each of them with different objectives. For example, Subdue (Cook and Holder 1994) is an algorithm that implements a computationally-constrained beam search, reducing the complexity. On the other hand, some algorithms reduce the computational complexity by imposing topological restrictions on the input graphs (Zaki 2001). However, other works explore strategies where the completeness is not sacrificed. For example AGM (Inokuchi and Motoda 2003), FSG (Kuramochi and Karypis 2002), SEuS (Shayan 2002), gSpan (Xifeng and Jiawei 2002) are examples where several heuristics and representations are explored, which aim to reduce the number of operations to perform.

In this research we present a new algorithm to detect the instances of a graph *G'* in a graph *G*, where a linear sequence of codes is used to represent the graphs (gSpan implemented this concept successfully). A code represents the information of an edge label and its adjacent vertices. Each code in a linear sequence is sorted with a new lexicographic order using criteria such as degrees, statistical summaries of the codes and a label order. The sequences of codes are built with a step by step expansion without candidate's generation. Also, we implement a width-depth search algorithm with a pruning phase that aims to reduce the number of operations to perform. This paper presents a detailed analysis of the proposed algorithm, being an extension of a previous work (Olmos et al 2004).

Definitions and Notation

The strategies presented in this work are focused in simple connected labeled graphs. In this section we define some concepts like graphs, subgraph and subgraph isomorphism.

A non directed simple graph G is a 6 – tuple $G = (V, E, L_V, L_E, \propto, \beta)$, where:

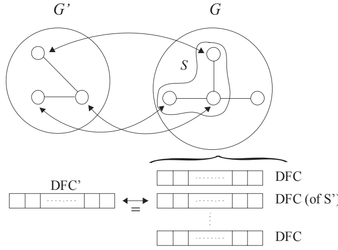
- $V = \{v_i \mid i = 1, \dots, n\}$, is the finite set of vertices, $V \neq \emptyset$
- $E \subseteq V \times V$, is the finite set of edges, $E = \{e = \{v_i, v_j\} \mid v_i, v_j \in V\}$
- L_V, is a set of vertex labels
- L_E, is a set of edge labels
- $\propto: V \rightarrow L_V$, is a function assigning labels to the vertices
- $\beta: E \rightarrow L_A$ a function assigning labels to the edges

A **subgraph** S of G, denoted by $S \subseteq G$, $S = (V^S, E^S, L_V^S, L_E^S, \propto^S, \beta^S)$ is a graph such that $V^S \subseteq V$, $E^S \subseteq E$, $L_V^S \subseteq L_V$, $L_E^S \subseteq L_E$.

Given two graphs $G' = (V', E', L'_V, L'_E, \propto', \beta')$, $G = (V, E, L_V, L_E, \propto, \beta)$, G' is **isomorphic** to G, denoted by $G' \cong G$, if there exist two functions f: $V' \rightarrow V$ and g: $E' \rightarrow E$ as bijections, where: (1) $\forall\ v' \in V'$, $\propto'(v') = \propto(f(v'))$ and (2) $\forall\ \{v'_i, v'_j\} \in E'$, $\beta'(\{v'_i, v'_j\}) = \beta(g(\{v'_i, v'_j\}))$.
Let G' and G be two graphs, we say that G' is a **subgraph isomorphic** of G if there exists $S \subseteq G$ such that $G' \cong S$.

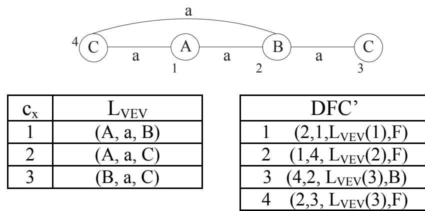
List Code Based Representation

Our goal consists on finding the instances of a graph G' in a graph G. We represent graphs through linear sequences of codes, where a code symbolizes an edge e, including its label and the labels of its adjacent vertices (a code based representation has been successfully applied in gSpan (Xifeng and Jiawei 2002)). Our method starts by building a model of G', represented by a linear sequence of codes, called DFC'. Based in DFC', the matching process tries to build a linear sequence DFC of G, where the corresponding code entries of DFC' and DFC must be identical. If DFC could be generated, then we found $S \subseteq G$, where S is represented by DFC and $G' \cong S$. This idea is



shown in Fig. 1 a) and Fig. 1 b) this is an example of a DFC code (the example is explained latter).

a)



c_x	L_{VEV}
1	(A, a, B)
2	(A, a, C)
3	(B, a, C)

DFC'
1 (2,1,L_{VEV}(1),F)
2 (1,4, L_{VEV}(2),F)
3 (4,2, L_{VEV}(3),B)
4 (2,3, L_{VEV}(3),F)

b)

Fig. 1. Example of a DFC Code

The DFC' model is a sorted sequence of codes $<dfc'_1, \ldots, dfc'_n>$, where each entry $dfc'_x = (i, j, c_k, t)$ and i, j, are the indexes associated to the adjacent vertices of edge $e_x \in E'$, c_k is a code that represents the label's information of the edge e_x, c_k is the L_{VEV} code associated to e_x and t is a special mark to classify the edges as F or B edges (their meaning is explained later on). To sort the codes dfc'_x, we consider the following rules.
First, we assume that there exists a linear order ($<_{V'}$) in the set L'_V and a linear order ($<_{E'}$) in the set L'_E. Based on ($<_{V'}$) and ($<_{E'}$), we derive a new linear order L_{VEV}, where the label's combinations of the vertices and edges are considered. Formally, $L_{VEV} = \{c_i \mid i = 1,\ldots, s\}$, s is the number of different combinations of labels $\propto'$, β' and $\propto'$ in G' where $\forall\ c_x = (\propto'(v_i), \beta'(e_a), \propto'(v_j))$, $c_y = (\propto'(v_m), \beta'(e_b), \propto'(v_n))$: $c_x \leq c_y$ if

- $\propto'(v_i) <_V \propto'(v_m)$ or
- $\propto'(v_i) =_V \propto'(v_m)$ and $\beta'(e_a) <_E \beta'(e_b)$ or
- $\propto'(v_i) =_V \propto'(v_m)$ and $\beta'(e_a) =_E \beta'(e_b)$ and $\propto'(v_j) \leq_V \propto'(v_n)$

L_{VEV} can be used to sort DFC', however L_{VEV} only considers the label's values. For this reason, we created another rule that includes the degree factor of the vertices.
The DFC' sequence is built to explore the edges in the graph with a Depth First Search strategy (DFS) (Thulasiraman 1992). DFS generates a spanning tree (T), inducing a partition in the set of edges of G' into *forward* (F) and *backward* (B) edges (T includes only forward edges). DFS imposes direction on the edges of G', which depends on the order of visit. Moreover, G' can be represented by more than one DFC', since the adjacent edges of a vertex may be chosen in arbitrary order and in the beginning, each vertex in G' can be selected as a root of T. In our case, G' is explored with a DFS strategy with some restrictions. These restrictions are focused to improve the mapping process as follows:

a) First, visit the vertex with the highest degree or,
b) Visit the vertex with the label that has the largest number of instances in the graph or
c) Visit the vertex based on the L_{VEV} order.

With these restrictions and using a DFS strategy, the DFC' code is built as follows:

1. Select a non visited vertex v_i in G' (v_i satisfying restrictions a), b) and c)
2. Expand vertex v_i to vertex v_j, (v_j has not been visited yet and satisfies restrictions a, b and c)
3. Add (i, j, $L_{VEV}(\{v_i, v_j\})$, F) to DFC' (a forward edge)
4. Expand all possible backward edges: $\forall\ \{v_j, v_x\}$ where v_x has already been visited, add (j, x, $L_{VEV}(\{v_j, v_x\})$, B) to DFC' according to the order L_{VEV}
5. If v_j has an adjacent vertex v_x that has not been visited yet, go to step 2 with v_j as v_i and v_x as v_j
6. If there are vertices that have not been visited yet, go to step 1. In other case, finish

DFC' includes all the edges in G', because we consider that it is a connected graph. Figure 1 b) shows an example of a L_{VEV} code. Note that this sequence has only 3 entries, which represent the different combinations of the labels in the graph. L_{VEV} is indexed once it was sorted. Then, the numerical value of the corresponding code L_{VEV} is stored in each dfc'_x. In this example, DFC' start with $dfc_1 = (2, 1, L_{VEV}(1), F)$ since v_2 is the vertex with the highest degree and, the expansion was directed to vertex v_1 because it is

the adjacent vertex to v_2 with the highest degree and the minor L_{VEV} code.

In the following section, we explain how the mapping between *G'* and *G* is constructed.

The Algorithm

With the DFC' model, the mapping process will be constrained to find a DFC code of *G* where |DFC'| = |DFC| and for every entry dfc'_x in DFC' and dfc_x in DFC, the L_{VEV} fields are the same. Then, if DFC is generated, we conclude that there exist $S \subseteq G$ where $G' \cong S$. We propose an algorithm divided in two phases: a pruning phase and a processing phase. The pseudocodes are shown in Fig 2 and Fig 3.

The pruning phase, step 1 of Fig 2, consists on removing $\forall v \in V$: $\alpha(v) \notin L'_V$ and $\forall e \in E$: $\beta(e) \notin L'_E$. This task can perform with the use of L_{VEV}, removing $\forall e \in G$ where $L_{VEV}(e) \notin L_{VEV}$. Moreover, with a statistical analysis based on L_{VEV}, we can remove edges in *G* without the minimum number of repetitions. We also use the vertices degrees and their number of repetitions in the graph to further prune *G*. Without loosing generality, consider that $deg_{min}(\alpha_1)$, …. ,$deg_{min}(\alpha_n)$ are the minimum degrees associated to the labels $\alpha_x \in L'_V$. Then, $\forall v \in V$ can not be common if $deg(\alpha(v)) < deg_{min}(\alpha')$ and $\alpha(v) = \alpha'$ (FSG uses a similar concept to order the vertices in an adjacency matrix). Note that after this preprocessing phase, *G* might have been partitioned in *s* connected graphs. Each of those graphs will be compared with *G'*. Let GSet be the sets of graphs derived from this process.

```
Input: (G', G)
Output: A mapping (if one exists)

1.  PreProcess(L_VEV, G, G', DFC', GSet)
2.  while GSet is not empty
3.      G_i ← select a graph from GSet and remove
             it in GSet
4.      S ← vertices in G_i mapping to dfc'_1(v_i)
5.      while S is not empty
6.        v ← select a vertex in S and
               S ← S - {v}
7.          ∀ e ∈ G_i: mark(e) = false
8.        R← RecursiveIteration(v,0,DFC',DFC,L_VEV)
9.        if R is true
             shows the mapping and finish
10.  G' is not a subgraph isomorphic of G
```

Fig. 2. The Mapping Main Routine.

The processing phase consists on finding the mapping between vertices and edges of *G'* and G_i, where $G_i \in$ GSet (the process is applied to each graph in GSet, while a DFC code has not been found, see Fig. 2). We use a backtracking algorithm, because this technique is fairly stable and performs well in most cases, since it does not require more resources than those strictly necessary and a new partial result is based on a previous result. Since DFC' is an array, the backtracking strategy can be used.

The process starts by finding the vertices *v* in G_i where $\propto(v) = \propto(v_i)$, $v_i \in G'$, $deg(v) \geq deg(v_i)$, and $dfc'_1 = (i, j, c_x, F)$ where $c_x = L_{VEV}(\{v_i, v_j\})$. Taking these vertices, the construction process of DFC begins, where each vertex forms a root of expansion. The vertices found are stored in S, where the vertices are sorted following rules a, b, and c described in the previous section.

Based on a selected vertex $v \in G_i$, the algorithm finds all the possible mappings that can be associated to entry dfc'_i of DFC', that is, each edge where the code c_x is the same. In a traditional DFS expansion, the rule is to expand in depth at any step. Nevertheless, in our case we implemented a width-depth search. The entries of DFC' are divided in two groups: *forward* (*F*) and *backward* (*B*) edges. If an entry is *forward*, it means that the edge is oriented to a vertex that is visited for the first time. In this case, we propose testing all the adjacent edges, that is, we test for all adjacent edges of a vertex *v'* in *G'* (*v'* is the vertex to expand) to see if there is an adjacent edge of vertex *v* in G_i, where *v* is the vertex associated to *v'* and their L_{VEV} codes are the same. All the mappings where there does not exist the minimum number of L_{VEV} combinations are pruned and we avoid exploring these combinations, line 8 in Fig 3.

```
RecursiveIteration(v, dim, DFC', DFC, L_VEV)
1.  if dim = |DFC'| return true
2.  dim ← dim +1
3.  (v'_i, v'_j, c_x, et) ← DFC'[dim]
4.  if not exist adjacent edges to v not visited
5.     v' ← vertex in DFC associated to v'_i
6.     return RecursiveIteration(v',dim-1, DFC',
                                    DFC,L_VEV)
7.  if et = "F"
8.   if Adj_List(v'_i)⊄Adj_List(v) (based on L_VEV
                                      codes)
9.     return false
10.  OpSet ← ∀ (v, v_k) ∈ Adj_List(v):L_VEV(v,v_k)=c_x
             and mark(v,v_k) = false
11.  while OpSet is not empty
12.    (v_i, v_j) ← select an entry in OpSet
13.    OpSet ← OpSet - (v_i, v_j)
14.    DFC[dim] ← (v_i, v_j)
15.    mark(v_i, v_j) = true
16.    if RecursiveIteration(v_j,dim,DFC',DFC,L_VEV)
          is true
17.       return true
18.    mark(v_i, v_j) = false
19. else
20.  if ∃ (v, v_j) where v_j is associated to v'_j
21.   DFC[dim] ← (v, v_j)
22.   mark(v, v_j) = true
22.   if RecursiveIteration(v_j,dim,DFC',DFC,L_VEV)
         is true
23.      return true
24.   else
25.      mark(v, v_j) = false
26.      return false
```

Fig. 3. RecursiveIteration Routine

On the other hand, backward edges are processed without special considerations, because if there is a mapping between edge *e*' in *G'* and edge *e* in G_i, then both of them have related adjacent vertices, lines 20 – 2 in Fig 3.

The input arguments of the RecursiveIteration function are: v, vertex to expand; dim, the dimension of the solution array; the DFC', sequence of codes of *G'*; DFC, the array solution and L_{VEV}. In the pseudocode of the Fig. 3, DFC only stored the indices of the vertices. However, we known the last two fields of the entries in DFC' are equal to the corresponding entries in DFC. Therefore, it is not necessary to store these values in the entries of DFC.

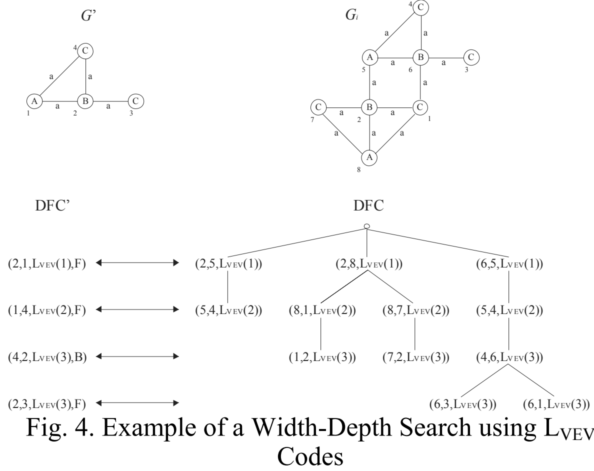


Fig. 4. Example of a Width-Depth Search using L_{VEV} Codes

Figure 4 shows an example based on the L_{VEV} code of figure 1. In this example, we can see how the partial result vectors are generated and pruned. Note that all the comparisons are done faster, because both dfc_x and dfc'_x have only numerical values. Moreover, it is possible to find all the instances of a graph *G'* in a graph *G*, just by leaving the algorithm run over all the possibilities. In this example, there are two mappings.

Analysis

In this section we demonstrate three important properties of the proposed algorithm: soundness, completeness, and termination guarantee.

Soundness

To prove that the algorithm is sound, it is necessary to prove the follows three properties.

Property 1. If DFC' = $<dfc'_1, \ldots, dfc'_n>$ and DFC = $<dfc_1, \ldots, dfc_m>$, then n=m.

Proof

It is easy to see that dim (from figure 3) stores the dimension of DFC, which is initialized as 0. If dim = |DFC'|, the process is stopped and |DFC'| = |DFC|, line 1 of Fig 3. Otherwise, for each successful recursive call, a new entry is added to DFC, where the entry comes from OpSet if it is a forward edge (line 14) or it searches for a backward edge (line 20). In both cases, dim increase in 1. Then, the algorithm builds a DFC sequence adding one entry for each successful recursive call, stopping the process when dim = |DFC'|.

Property 2. $\forall$ $dfc'_x \in$ DFC' and $dfc_x \in$ DFC, the L_{VEV} code is equal for both of them.

Proof

For each successful recursive call, a dfc_x entry is added to DFC. Note that dfc_x comes from OpSet, where $\forall$ $e \in$ OpSet, $L_{VEV}(e) = c_x$ (c_x corresponds to the L_{VEV} code of dfc'_x), line 10. Therefore, any entry that is selected from OpSet has the c_x code and then, dfc_x and dfc'_x have the same L_{VEV} code.

Property 3. DFC represents a valid mapping if there exists a function *f*: DFC' $\rightarrow$ DFC as a bijection where $f(dfc'_x) = dfc_x$.

Proof

It is easy to see that OpSet includes only edges *e* where mark(*e*) = false, line 10. That is, we only consider edges that have not been visited yet. Therefore, each dfc_x entry that is added to DFC represents a non visited edge and the corresponding edge *e* represented by dfc_x is not considered in futures recursive calls (in depth). Then, dfc_x will be associated to only one entry in DFC.

Note that the properties 1, 2 and 3 proofs that our algorithm is sound.

Completeness

To prove that the algorithm is complete, we must show that the pruning phase does not discard any valid mapping.

First, we prove that the algorithm is complete without any pruning. It is easy to see that the main routine considers every possible root in *G*, where the vertex label's are equal to the first vertex in DFC', line 4. On the other hand, the RecursiveIteration routine expands DFC adding a new entry, where this entry comes from OpSet. In this case, OpSet includes any adjacent edge to *v*, that is, the vertex in expansion. In other words, in each recursive call, every possible edge will be considered as a possible expansion path and then, in the worst case, the algorithm will explore all the possible combinations of codes (as a naive approach would do).

Now, we prove that the implemented pruning does not discard any valid mapping.

First, we start with the pruning phase. In this phase we first remove $\forall$ $v \in V$: $\alpha(v) \notin L'_V$ and $\forall$ $e \in E$: $\beta(e) \notin L'_E$. We use the L_{VEV} to remove $\forall$ $e \in G$ where $L_{VEV}(e) \notin L_{VEV}$. Note that if $e = (v_i, v_j)$ and $L_{VEV}(e) \notin L_{VEV}$, then $\alpha(v_i) \notin L'_V$ or, $\beta(e) \notin L'_E$ or, $\alpha(v_j) \notin L'_V$. Therefore, any valid mapping is affected.

We also implemented a quantitative analysis based on L_{VEV}, where we removed edges in *G* without the minimum number of repetitions. That is, $\forall$ $e \in G$ is removed if the number of edges in *G* with the same $L_{VEV}(e)$ is *k* and *k* is smaller than the number of edges in *G'* with the same $L_{VEV}(e)$ code.

Finally, in the pruning phase we use the degree of the vertices to remove $\forall\ v \in G$: $deg(\alpha(v)) < deg_{min}(\alpha')$ and $\alpha(v) = \alpha'$. So, if a vertex v is removed, it means that the number of edges incident to v is smaller than $deg_{min}(\alpha')$ where $\alpha(v) = \alpha'$ and then, any valid mapping will contain v (each time we remove a vertex v, a new quantitative analysis is performed to identify another vertex v to remove).

Besides the previous pruning phase, we perform a different pruning process inside RecursiveIteration, line 8. In this step, the list of adjacent edges of v'_i (the vertex from G' in expansion) is tested, searching that for each adjacent edge of v'_i, there exists at least one edge e in G to which it can be mapped, where e is adjacent to v. This process is performed based on L_{VEV} codes. If one adjacent edge of v'_i can not be mapped to its any edge in *OpSet*, it means that v'_i can not be mapped to v. Then, for each recursive call, vertices that do not share these property are discarded.

Termination Guarantee

To prove that the algorithm will always finish, we must show that the pruning and processing phases terminate (they will not run for ever).

The pruning phase removes vertices and edges from G that cannot generate valid mappings. In each step, if a vertex is removed, then we get a new graph $\hat{G}$, where #vertices in $\hat{G}$ is smaller than #vertices in G. Then, in the worst case, we need $|V|$ iterations to remove all vertices in G, and the process is stopped. A similar process happens if the edges are removed. Then, the pruning phase always terminates when $\hat{G}$ is empty.

On the other hand, the processing phase starts finding all vertices v in G where $\alpha(v) = \alpha(v')$ and v' is the first vertex considered by DFC'. Every vertex v is a possible root of expansion, and after we run RecursiveIteration with v as root, we known if vertex v is a valid root. So, if we prove that RecursiveIteration terminates for every root v, then we conclude that our algorithm terminates.

For each recursive call, the RecursiveIteration function builds an OpSet. Then, one edge $e \in$ OpSet is selected as expansion path, where its corresponding entry in DFC is added and the function calls itself. If the selected edge e does not generate a valid DFC code, then e is removed from OpSet and the process continues with a new edge e in OpSet until OpSet is empty.

Note that OpSet is finite (in the worst case, |OpSet| = #edges in G_i). Moreover, let k and k+1 be two successful recursive calls, where $OpSet_k$ and $OpSet_{k+1}$ are the sets generated in the recursive calls k and k+1 respectively. Then, if an edge $e \in OpSet_k$ is selected as an expansion path, $e \notin OpSet_{k+1}$, because e was marked as visited (line 15 of fig 3). Then, if dim < |DFC'|, the current recursive call builds OpSet without considering the edges included in the current DFC. Finally, as each OpSet is a subset of Adj_List(v) (line 10 of fig 3), then every OpSet generated in every recursive call is finite.

Since all the generated OpSet's determine the total number of recursive calls, we conclude that the algorithm, in the worst case, finishes (all the combinations between the edges in the OpSet's were tested).

With the above mentioned, we prove that the algorithm terminates.

Experimental Results

The performance of the proposed algorithm was examined using graphs variable number of vertices and edges. In our tests, G' maintains a proportion of 15% to 60% based on the number of vertices of G. Our experiment was divided in three sections: first, where all labels of the vertices and edges are different; second, all the labels are equal and third, tests where a subgraph isomorphic does not exist. All the experiments were done with a 2.8 Ghz Intel PC with 1 GB of main memory, running Red Hat Linux 9.0. All the reported times are expressed in seconds.

We compared the performance of our algorithm with an implementation available in the Web as part of the Subdue system (http://ailab.uta.edu/subdue/), called *sgiso*, where an expansion with candidate generation is employed. On the other hand, algorithms like AGM, FSG, SEuS, gSpan are not comparable directly, because these find common patterns in a set of graphs (moreover, some of them are not available and other versions have restrictions over the input graphs). However, concepts such as lists and summaries of codes, expansion without candidate generation are employed successfully in our approach.

G'		G		Sgiso Time	S.I.D. Time	Total Time.
$\lvert V'\rvert$	$\lvert E'\rvert$	$\lvert V\rvert$	$\lvert E\rvert$			
705	4290	1000	8047	1.78	0.01	1.44
750	4558	1000	8047	1.99	0.01	1.83
800	6431	1000	8047	3.09	0.03	4.47
851	6841	1000	8047	3.19	0.04	5.84
909	7303	1001	8047	3.52	0.05	6.97
952	7655	1000	8047	3.69	0.06	7.89
1000	10039	2000	20078	15.22	0.09	15.55
1000	10039	4000	40156	29.36	0.09	15.50
1000	10039	8000	80312	36.58	1	15.55

Table 1. Results of the Tests where all the Labels are Different

G'		G		Sgiso Time	S.I.D. Time	Total Time.
$\lvert V'\rvert$	$\lvert E'\rvert$	$\lvert V\rvert$	$\lvert E\rvert$			
120	177	601	900	-	0	0.03
240	357	600	900	-	0	0.03
360	537	609	900	-	0	0.03
159	474	805	2400	-	0	0.17
321	954	803	2400	-	0.01	0.18
480	1434	800	2400	-	0.01	0.21
204	877	1003	4500	-	0.01	0.56
400	1777	1007	4500	-	0.01	0.59
609	2676	1002	4500	-	0.04	0.70

Table 2. Results of the Tests where all the Labels are the Same

If all the labels are different, we know that the Subgraph Isomorphism Problem can be solved in P. According to our results shown (shown in Table 1), the response time of our algorithm suffers a proportional increase according to the dimension of the graphs. We registered the time taken to find S (Subgraph Isomorphism Detection Time, S.I.D.) and the total time taken to find S including the

preprocessing and the time to writing the output. Note that if the graph dimension of *G'* and *G* is small, *sgiso* reports similar runtimes (in some cases, better). This result can be explained because the structure of the graphs helps *sgiso* to find the right mapping. However, if the graph dimension is larger, our representation and the pruning phase show their effectiveness with shorter runtimes.

Table 2 shows the results of the experiments where all the labels of the vertices and edges are the same. For this case, *sgiso* was not able to find a solution in less than 2 hours (represented by "-"). This behaviour is consequence of the input data, because the number of combinations increases a lot. On the other hand, our approach has a better response time. It is a consequence of our heuristics, because not all vertices have the same L_{VEV} combination and degree.

Finally, Table 3 shows some results where there does not exist any instance *S* in *G* (all labels are different). These experiments show that our algorithm found a solution very quickly, because our pre-processing phase prunes a high number of vertices and edges of the graph (in our experiments the *G* graphs were discomposed in subgraphs with only one vertex). On the other hand, *sgiso* had to do a larger number of operations to discover that there did not exist any mapping.

Our results show that our algorithm adjusts to the patterns in the graphs. On the other hand, the reduction of time between *sgiso* and our approach is a consequence of the representation proposed together with our step by step expansion without candidate generation.

G'		G		Sgiso	S.I.D.	Total
$\|V'\|$	$\|E'\|$	$\|V\|$	$\|E\|$	Time	Time	Time.
120	1482	605	35943	7.39	0	0.13
120	1445	595	35946	7.38	0	0.12
120	1471	600	95942	7.47	0	0.12
160	2644	806	63923	24.26	0	0.34
160	2639	809	63921	24.29	0	0.38
160	2547	800	63922	24.50	0	0.26
200	2022	1020	64824	59.47	0	0.69
200	2062	1011	83441	60.01	0	0.74
200	2011	1015	83475	60.40	0	0.68

Table 3. Results of the Tests where no Mapping Exists

Conclusions and Future Work

In this work we presented a new approach to solve the Subgraph Isomorphism Problem. We implement concepts like a list code based representation, a step by step expansion model without candidate generation, a width-depth search and a prune phase. In our experiments, we used several types of graphs with the aims to get a good measure of the proposed algorithm performance. Our experiments showed that our approach prunes a high number of paths and finds the mappings very quickly. As result, the global performance is attractive. We will continue our research testing our approach with other algorithms focused to solve the problem. We will also do experiments with real domains, as the chemical toxicity and Web logs domains. We are also studying the possibility to implement this approach in the Subdue system, where it is necessary to evaluate when a graph is a subgraph isomorphic to another graph. Finally we want to extend our algorithm to solve problems such as graph based data mining.

References

Cook, D. J.; Holder, L. B. Substructure discovery using minimum description length and background knowledge. Journal of Artificial Intelligence Research, 1:231-255, 1994.

Garey R. Michael. Johnson S. David. Computers and Intractability, A Guide to the theory of NP-Completeness. W. H. Freeman and Company (2003)

Inokuchi, A. Washio, T. Motoda, H. Complete Mining of Frequent Patterns from Graphs: Mining Graph Data. Machine Learning, Kluwer Academic Publishers. (2003). 321 – 354

Kuramochi, M.; Karypis, G. An Efficient Algorithm for discovering Frequent Subgraphs. Tech. Report. Dept. of Computing Science, University of Minnesota. June (2002).

Olmos, Ivan. Gonzalez, Jesus. Osorio, Mauricio. Subgraph Isomorphism Detection with a Representation Based on List of Codes. First Iberoamerican Workshop on Machine Learning for Scientific Data Analysis (2004).

Shayan G. Sudarshan S.C. "SEuS: Structure Extraction using Summaries". Technical Report. C. S. Department, University of Maryland, (2002).

Thulasiraman, K. Swamy, M.N.S. Graphs: Theory and Algorithms. Wiley – Interscience. 460 pp (1992).

Xifeng, Y. Jiawei H. gSpan: Graph – Based Substructure Pattern Mining. Technical Report. University of Illinois. (2002).

Zaki, M. J. Efficiently Mining Trees in a Forest. Tech. Report 01-7, Computer Science Department, Rensselaer Polytechnic Institute, (2001).