DIGDAG, a first algorithm to mine closed frequent embedded sub-DAGs

Alexandre Termier
Institute of Statistical Mathematics
Tokyo, Japan

Yoshinori Tamada
Systems Pharmacology Research Institute, GNI Ltd
Fukuoka, Japan

Kazuyuki Numata
Institute of Medical Sciences
University of Tokyo, Japan

Seiya Imoto
Institute of Medical Sciences
University of Tokyo, Japan

Takashi Washio
I.S.I.R.
University of Osaka, Japan

Tomoyuki Higuchi
Institute of Statistical Mathematics
Tokyo, Japan

Abstract

Although tree and graph mining have attracted a lot of attention, there are nearly no algorithms devoted to DAG-mining, whereas many applications are in dire need of such algorithms. We present in this paper DIGDAG*, the first algorithm capable of mining closed frequent embedded sub-DAGs. This algorithm combines efficient closed frequent itemset algorithms with novel techniques in order to scale up to complex input data.*

1. Introduction

A growing percentage of the data available today is *semi-structured data*, i.e. data that can be represented as a labeled graph. This kind of data can be found in various domains such as bioinformatics, chemistry or computer networks. To analyze this data, there is an important need of specific data mining algorithms, especially for the discovery of frequent patterns. In the recent years, the research community has produced numerous algorithms to find frequent patterns in general graphs [5] and in trees [2, 7, 10]. However, there are nearly no algorithms for finding frequent patterns in data structured as DAGs. This is a problem to be explored, as a lot of interesting data exhibit this structure. An important body of this data is the result of Bayesian networks based algorithms. This is for example the case of most current studies about gene interaction networks [4]. Such data exhibit large structural variations, meaning that ancestor-descendant relationships are more often preserved than parent-child relationships. Hence current DAG mining [1] and graph mining [5, 9] algorithms, which discover *induced patterns* based only on parent-descendant relationship, are not adapted to analyze this data. Algorithms discovering *embedded patterns*, based on the ancestor-descendant relationships, are needed for this problem.

In this paper, we present DIGDAG, the first algorithm capable of mining closed frequent embedded sub-DAGs (*c.f.e.-DAGs*) in DAG data. The input data are DAGs where all the labels must be distinct. This assumption of distinct labels has long lead to think that the solution to this problem was trivial. However, we will show that naive approaches are unable to provide results on real data and this motivates the more elaborate method used in DIGDAG. Experiments will show that DIGDAG significantly improves a naive approach both in term of memory consumption and in term of computation time.

The paper is organized as follows: Section 2 gives the necessary definitions, Section 3 presents the DIGDAG algorithm, Section 4 shows some preliminary experiments, and Section 5 concludes the paper and gives perspectives for future research.

2. Definitions and data-mining problem

A **labeled graph** is a tuple $G = (V, E, \varphi)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $\varphi : V \mapsto \mathcal{L}$ is a labeling function with $\mathcal{L}$ a finite set of labels. For an edge $(u, v) \in E$, u is the **parent** of v and v is the **child** of u. If there is a set of vertices $\{u_1, ..., u_n\} \subseteq V$ such that $(u_1, u_2) \in E$,...,$(u_{n-1}, u_n) \in E$, $\{u_1, ..., u_n\}$ is called a **path**, u_1 is an **ancestor** of u_n and u_n is a **descendant** of u_1. There is a **cycle** in the graph if a path can be found from a vertex to itself. An edge $(u, v) \in E$ of the graph is said

to be a **transitive edge** if besides the edge (u, v), there also exists another path from u to v in G. A **labeled DAG** is a labeled graph without cycles.

Let $P_1 = (V_1, E_1, \varphi_1)$ and $P_2 = (V_2, E_2, \varphi_2)$ be two DAGs. P_1 is an **embedded** sub-DAG of P_2, written $P_1 \sqsubseteq P_2$, if there exists an injective homomorphism $\mu : P_1 \mapsto P_2$ such as: 1) for two vertices $u \in V_1$ and $v \in V_2$ such that $v = \mu(u)$, $\varphi_2(v) = \varphi_1(u)$ holds and 2) for $(u, v) \in E_1$, $\mu(u)$ is an ancestor of $\mu(v)$ in P_2. In short, the homomorphism must preserve the labels and the ancestor relationship. If in 2), the homomorphism only preserves the parent relationship (i.e. the condition becomes: for $(u, v) \in E_1$, $\mu(u)$ is a parent of $\mu(v)$ in P_2), then P_1 is an **induced** sub-DAG of P_2. Note that an induced sub-DAG is also an embedded sub-DAG, but not the opposite.

Let $\mathcal{D} = \{D_1, ..., D_n\}$ be a set of labeled DAGs and $\varepsilon \geq 0$ be an absolute frequency threshold. A DAG P is a **frequent embedded (*induced*)** sub-DAG of $\mathcal{D}$ if it is embedded (*induced*) in at least ε DAGs of $\mathcal{D}$. The set of DAGs of $\mathcal{D}$ in which P is embedded (*induced*) is called the **tidlist** of P, denoted $tidlist(P)$. The **support** of P is defined by $support(P) = |tidlist(P)|$.

A frequent embedded (*induced*) sub-DAG P of $\mathcal{D}$ is **closed** if it is maximal for its support, i.e. if there is no frequent embedded (*induced*) sub-DAG P' of $\mathcal{D}$ such that P is embedded (*induced*) into P' and $tidlist(P) = tidlist(P')$.

The exact data mining problem at hand is to find closed frequent embedded sub-DAGs (c.f.e.-DAGs) in a collection of DAG data, where for each input DAG all its labels are distinct.

3. Algorithm

3.1. Overview

We present in this section DIGDAG, an efficient algorithm for discovering c.f.e.-DAGs in DAG data where for each input DAG, all the labels are distinct. The fact that all the labels are distinct in the input DAGs is convenient because it means that there are no ambiguities, i.e., if two edges (A, B) and (B, C) are found to appear frequently together in the input DAGs $\{D_1, ..., D_p\}$, then their composed pattern $A \rightarrow B \rightarrow C$ is necessarily frequent and appears in $\{D_1, ..., D_p\}$. This can be further extended, and any c.f.e.-DAG P is uniquely defined by its set of edges E_P.

This means that instead of mining directly c.f.e.-DAGs, the c.f.e.-DAGs are mined by deriving closed frequent sets of edges. Though there are no algorithms to mine c.f.e.-DAGs, a lot of closed frequent itemset mining algorithms, capable of mining efficiently sets of any kind of items [6, 8], can then be used to mine the c.f.e.-DAGs.

As we are interested in finding closed frequent *embedded* sub-DAGs, the ancestor-descendant relations among vertices must be introduced as input data. For each DAG D we already have its set of edges E_D. We define the saturated set $E_D^+ = \{(u, v) \mid u \text{ is an ancestor of } v \text{ in } D\}$ containing all the ancestor-descendant edges of a DAG D. Each input DAG $D \in \mathcal{D}$ will be represented by its E_D^+ set. The set $\mathcal{D}^+ = \{E_D^+ \mid D \in \mathcal{D}\}$ regroups all the saturated versions of the input DAGs.

Now the question is how to correctly use a closed frequent itemset mining algorithm to discover the sets of edges of the c.f.e.-DAGs. Briefly stated, it can be summed up as a two step process.

1. Using $\mathcal{D}^+$ as input, apply a closed frequent itemset mining algorithm to discover closed frequent ancestor-descendant edge sets.

2. Process the closed frequent ancestor-descendant edge sets to discover c.f.e.-DAGs.

Both of these two steps seem quite straightforward and very easy to implement, i.e., step 1 should be the simple application of any well known closed frequent itemset mining algorithm, and step 2 should only be connecting the edges together to discover the c.f.e.-DAGs. However this simplicity is only apparent, and in practice numerous problems arise, which prevent any "naive" approach to success. The main of these problems is that step 1 doesn't scale up to complex data. We have tried to analyze real data made of 100 DAGs, each having 300 vertices, coming from a bioinformatics problem. The closed frequent itemset algorithm was the FIMI 2003 winner, LCM2 [8]. This algorithm uses a lot of techniques to reduce the size of the dataset in memory. But even on a machine with 8 GB of RAM, the algorithm could not complete due to memory saturation. This is due to two reasons. First, the item space is the space of all the ancestor-descendant edges, which is huge. This is difficult for any closed frequent itemset mining algorithm. But the main reason lies in the very nature of the closed frequent ancestor-descendant edge sets researched. In the ideal case, ancestor-descendant edges are regrouped together because they belong to the same c.f.e.-DAG. But two c.f.e.-DAGs can also appear frequently together in any $D \in \mathcal{D}$, in this case all their ancestor-descendant edges will be regrouped in the same closed frequent ancestor-descendant edge set. So in facts, the closed frequent itemset algorithm does not look for the c.f.e.-DAGs, but it looks for all the *closed frequent sets of c.f.e.-DAGs*. This problem is much more difficult than the original problem, and in complex real data the sheer number of possible c.f.e.-DAGs combinations gives too many results for an efficient handling in memory.

In the following, we will explain the methods we developed to avoid these problems in the DIGDAG algorithm[1].

[1]NB: We will always refer to the vertices of the DAGs simply by their labels.

3.2. Item space reduction using the tiles

Experiments show that the successful use of the ancestor-descendant edges as items for a closed frequent itemset algorithm is not easy on real data, because the item space they make is too big for efficient handling. Our solution to reduce the item space size is to use instead special groups of ancestor-descendant edges named *tiles* as items. The tiles are the closed frequent sets of ancestor-descendant edges that share the same initial label, like $\{(a, b_1), ..., (a, b_k)\}$ $(a, b_1, ..., b_k \in \mathcal{L})$ for example. They can be represented as depth 1 trees. Lets $\mathcal{T}$ denote the set of all tiles. Figure 1 shows a simple example of tile discovery. The input DAGs are D_1 and D_2. For each label a matrix is constructed, whose lines are the vertices of the given label and whose columns indicate the labels of the descendants of these vertices. For example for label A there are two lines corresponding to the vertices A_1 and A_6. A_1 has $\{C, D, E\}$ as descendants, whereas A_6 has $\{D, E, F, H\}$ as descendants. Applying a closed frequent itemset mining algorithm on this matrix gives the tiles whose root has the given label, as shown on the right of the figure. Note that in the figure, matrices for labels A, B and D only are shown, other labels, having no descendants, do not produce tiles.

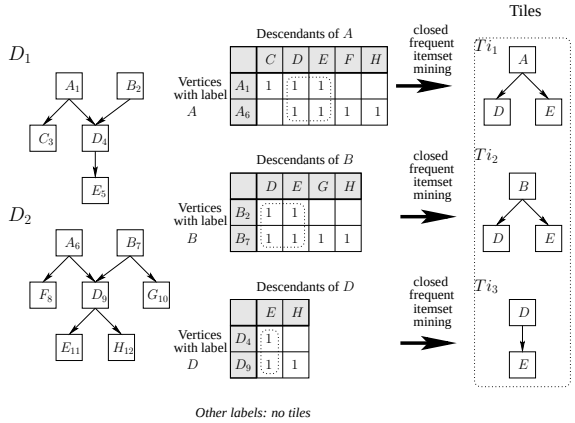

Figure 1: Example of tile discovery, frequency threshold $\varepsilon = 2$. Vertex identifiers are subscripts of vertex labels.

3.3. Further item space reduction by tiles grouping

The input DAGs can be represented by the tiles they contain. When, for each DAG $D \in \mathcal{D}$, Ti_D is the set of tiles included in D, the new input data is $\mathcal{D}^{Ti} = \{Ti_D \mid D \in \mathcal{D}\}$. Applying a closed frequent itemset mining algorithm on $\mathcal{D}^{Ti}$ gives closed frequent tilesets, from which the c.f.e.-DAGs can be deduced. However in this case also the results are closed frequent sets of c.f.e.-DAGs, which are too numerous for successful computation in complex cases.

Our solution to this combinatorial problem is to make computations where the item space is reduced to small subsets of the whole tile set $\mathcal{T}$. The ideas are that each c.f.e.-DAG is likely to contain only a few tiles of $\mathcal{T}$, and that two disconnected c.f.e.-DAGs that appear frequently in the same input DAGs will necessarily contain different tiles (thanks to the distinct labels hypothesis). A further observation is that given D a c.f.e.-DAG and l a vertex of D, let T_l be the biggest tile included in D whose root is l. Then the leaves of T_l represent all the descendants of l in D. Let l' be one of these descendants, then the set of all the vertices of $T_{l'}$, the biggest tile of root l' included in D, will be included in the set of leaves of T_l.

We have used this monotonic property to create an simple heuristic method grouping the tiles based on their labels. The result is a set of tile groups $\mathcal{TG}$, satisfying the following properties[2].

Property 1 *For any c.f.e.-DAG D, there exists a tile group $TG \in \mathcal{TG}$ such that TG contains all the tiles contained in D.*

Property 2 *For all $TG \in \mathcal{TG}$, TG is unlikely to contain the tiles of two c.f.e.-DAGs D_1 and D_2 that appear frequently in the same DAGs, even if this can happen in rare cases.*

For each tile group $TG \in \mathcal{TG}$ we perform a closed frequent itemset algorithm on $\mathcal{D}^{Ti}_{|TG}$, i.e. $\mathcal{D}^{Ti}$ reduced to the tiles of TG. A closed frequent tileset obtained here correspond to the tiles of an ancestor-descendant relationship saturated c.f.e.-DAG. Even if a closed frequent tileset contains multiple c.f.e.-DAGs, a simple safety check is performed to further decompose it into each of the c.f.e.-DAGs it contains.

3.4. Discovering the c.f.e.-DAGs from their tilesets

Having determined each tileset that corresponds exactly to the tiles of an ancestor-descendant relationship saturated c.f.e.-DAG D^+, we know that for each c.f.e.-DAG D we have all its ancestor-descendant edges, i.e. we have E^+_D. But two different c.f.e.-DAGs D_1 and D_2 can be such that $E_{D_1} \neq E_{D_2}$, but $E^+_{D_1} = E^+_{D_2}$. We need further processing to find the non-saturated c.f.e.-DAGs, which are the expected final result. For this we distinguish the edges of a c.f.e.-DAG into two categories, i.e., the *direct edges* and the *transitive edges*. An edge (u, v) is a direct edge if the only way from u to v in the c.f.e.-DAG is the edge (u, v). And edge (u, v) is a transitive edge if it is not a direct edge, i.e. if from u to v there are several paths. These are shown in Figure 2.

[2]Proofs are omitted by lack of space

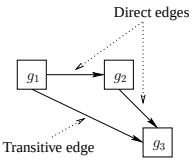

Figure 2: Simple example of directs and transitive edges

The direct edges of the c.f.e.-DAGs are easily found from the saturated c.f.e.-DAGs. The transitive edges are the tricky part. For every transitive edge of each saturated c.f.e.-DAG D^+, we check if the transitive edge is actually contained in all the input DAGs containing D^+. This check is efficiently done by applying a closed frequent itemset mining algorithm to the edge sets of all the input DAGs containing D^+. An edge of each closed frequent edges set found here represents a transitive edge of a non saturated c.f.e.-DAG in the case where the edge is not a direct edge of D^+.

We have proved the following property.

Property 3 *The set of DAGs found by* DIGDAG *is the sound and complete set of the c.f.e.-DAGs.*

4. Experiments

This section is divided into two parts. First, we show that the techniques used in the DIGDAG algorithm drastically reduce computation time and memory usage over the naive method of the beginning of Section 3. Then, we give an application example with the analysis of the action of the *terbinafine* drug.

The dataset used is a real world dataset of gene network DAGs, as outputted by the algorithm of [4] when analyzing microarray data from [3]. This algorithm models gene networks with Bayesian networks. From the input microarray data, a greedy search is performed, whose results are local optima, the *candidate gene networks*. They are of mixed quality: they contain both correct and incorrect parts. The goal of our data mining algorithm is to extract closed frequent gene networks patterns from the candidate gene networks, which should be more credible and more easy to analyze than the whole candidate networks.

To keep reasonable runtimes for all algorithms, we used a simplified dataset reduced to 100 candidate gene networks of 50 genes. The DAGs of this dataset have in average 284.5 ancestor-descendant edges, with a maximum of 352 ancestor-descendant edges.

The machine used is a Xeon 2.8 GHz with 8 Gb of RAM running under Linux. The implementations of DIGDAG and of the naive algorithm are our own C++ implementation, using inside the LCM2 closed frequent itemset mining algorithm[3].

[3]http://research.nii.ac.jp/∼uno/codes.htm

4.1. Comparison with the naive method

We have presented in Section 3 a naive method for extracting closed frequent embedded DAGs, and claimed that this method was not efficient enough for real data analysis. We have implemented this method in a way very similar to DIGDAG, and show the performance difference with DIGDAG in Figures 3 and 4.

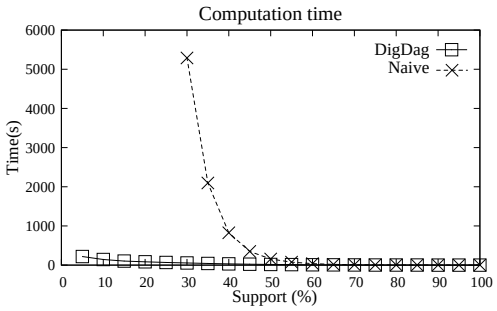

Figure 3: Naive vs DIGDAG, computation time

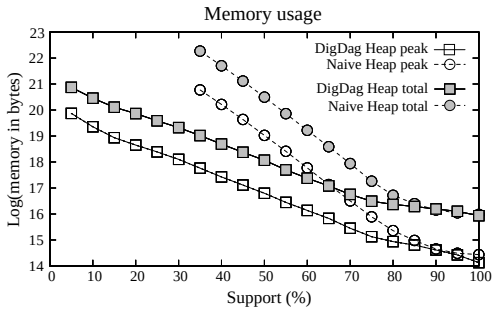

Figure 4: Naive vs DIGDAG, memory usage

Figure 3 shows the difference in computation time. DIGDAG is at least two orders of magnitude faster than the naive method: the improvement made is significant. As a consequence, DIGDAG can find results quickly for low support values, while the naive method's computation time exceeds the allocated 2 hours for support values below 30%.

The memory consumption difference is shown on Figure 4. For each program we give two values, as reported by the Linux $memusage$ command: the *heap total*, which corresponds to the total amount of memory allocated, and the *heap peak*, which corresponds to the maximum amount of system memory used by the program during its execution. Due to problems with the $memusage$ command, this experiment was made with a machine different from the previous one, a dual-core Xeon at 2.2 GHz with 4 Gb of RAM running under Linux. The naive method saturates the available memory for support values under 35%. For the lowest support value, the figure shows that the naive method consumes much more memory than DIGDAG: for support value

35%, the heap peak of DIGDAG is three orders of magnitude lower than the heap peak of the naive method. Since a support value of 65%, the heap peak of the naive method is even higher than the heap total of DIGDAG. These results shows that DIGDAG is much less prone to saturating the system's memory during computation. This is very important; as if the memory is saturated the program will not give any results, and computation time will have been wasted.

From these experiments, we can see that the improvements made by DIGDAG over the naive method are indeed efficient at reducing the memory consumption of the program. These improvements also allow an important gain in computation time.

4.2. Application

In the previous dataset, the 50 genes have not been chosen at random, they have been carefully selected as being the most probably affected by the *terbinafine* drug, in order to study the action of this drug.

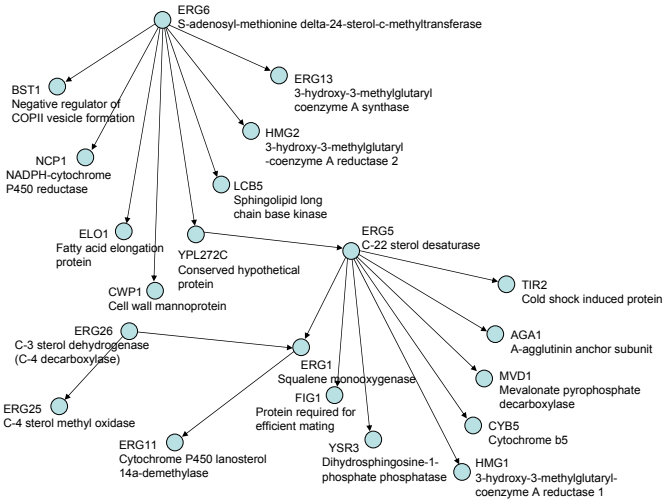

Figure 5: A significative c.f.e.-DAG.

One of the c.f.e.-DAGs found is shown on Figure 5. This gene network clearly shows relations of the gene ERG1 (the real target of terbinafine) with the genes ERG6, 11, 13, 25 and 26 from the steroid metabolism, and HMG1 and 2 from the lipid metabolism. This suggests that terbinafine works on steroid metabolism or lipid metabolism related genes, which is consistent with existent knowledge about this drug. Our c.f.e.-DAG proposes a precise interaction structure between these genes, which could serve as a basis to design verification experiments.

5. Conclusion and future works

We have presented in this paper the DIGDAG algorithm, the first algorithm capable of mining c.f.e.-DAGs. Our first experiments on real data have shown that this algorithm exhibited much better mining performances than a naive approach for the task at hand, and can allow to find interesting patterns.

As a topic for future research, we would like to be able to handle even more complex data (more vertices and more input DAGs) through the use of parallelism. We would also like to research what modifications would be needed for DIGDAG to handle general graphs instead of DAGs.

Acknowledgements: This research was, in part, supported by Function and Induction project of ROIS/TRIC.

References

[1] Y.-L. Chen, H.-P. Kao, and M.-T. Ko. Mining dag patterns from dag databases. In *Web-Age Information Management (WAIM), Dalian, China*, 2004.

[2] Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, 2004.

[3] T. R. Hughes, M. J. Marton, A. R. Jones, C. J. Roberts, R. Stoughton, C. D. Armour, H. A. Bennett, E. Coffey, H. Dai, Y. D. He, M. J. Kidd, A. M. King, M. R. Meyer, D. Slade, P. Y. Lum, S. B. Stepaniants, D. D. Shoemaker, D. Gachotte, K. Chakraburtty, J. Simon, M. Bard, and S. H. Friend. Functional discovery via a compendium of expression profiles. *Cell*, 102(1):109–126, July 2000.

[4] S. Imoto, T. Goto, and S. Miyano. Estimation of genetic networks and functional structures between genes by using bayesian network and nonparametric regression. In *Pacific Symposium on Biocomputing*, pages 175–186, 2002.

[5] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

[6] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel*, 1999.

[7] A. Termier, M. Rousset, and M. Sebag. Dryade : a new approach for discovering closed frequent trees in heterogeneous tree databases. In *International Conference on Data Mining ICDM'04, Brighton, England*, pages 543–546, 2004.

[8] T. Uno, M. Kiyomi, and H. Arimura. Lcm v.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *2nd Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, 2004.

[9] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD*, pages 286–295, 2003.

[10] M. J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 65(1-2):33–52, March/April 2005.