First publ. in: Encyclopedia of Data Warehousing and Mining, 2. ed. - Hershey, PA, USA : IGI Global, 2008, pp. 1865-1870

Subgraph Mining

Dr. Ingrid Fischer*
ALTANA Chair for Bioinformatics and Information Mining
Department of Computer and Information Science
University of Konstanz
Box M712
78457 Konstanz
Germany

voice: +49 9131 85-27830
fax: +49 9131-85-28809
email: fischer@inf.uni-konstanz.de

Thorsten Meinl
ALTANA Chair for Bioinformatics and Information Mining
Department of Computer and Information Science
University of Konstanz
Box M712
78457 Konstanz
Germany

voice: +49 7531 88-5016
fax: +49 7531-88-
5132
email: meinl@inf.uni-konstanz.de

(* Corresponding author)

Subgraph Mining

Ingrid Fischer and Thorsten Meinl

University of Konstanz, Germany

INTRODUCTION

The amount of available data is increasing very fast. With this data, the desire for data mining is also growing. More and larger databases have to be searched to find interesting (and frequent) elements and connections between them. Most often the data of interest is very complex. It is common to model complex data with the help of graphs consisting of nodes and edges that are often labeled to store additional information. Having a graph database, the main goal is to find connections and similarities between its graphs. Based on these connections and similarities, the graphs can be categorized, clustered or changed according to the application area. Regularly occurring patterns in the form of subgraphs — called *fragments* in this context — that appear at least in a certain percentage of graphs, are a common method to analyze graph databases. The actual occurrence of a fragment in a database graph is called *embedding*. Finding the fragments and their embeddings is the goal of subgraph mining described in detail in this chapter.

The first published graph mining algorithm, called Subdue, appeared in the mid-1990s and is still used in different application areas and was extended in several ways. (Cook & Holder, 2000). Subdue is based on a heuristic search and does not find all

possible fragments and embeddings. It took a few more years before more and faster approaches appeared. In (Helma, Kramer, & de Raedt, 2002) graph databases are mined for simple paths, for a lot of other applications only trees are of interest (Rückert & Kramer, 2004). Also Inductive Logic Programming (Finn et al., 1998) was applied in this area. At the beginning of the new millennium finally more and more and every time faster approaches for general mining of graph databases were developed that were able to find all possible fragments. (Borgelt & Berthold, 2002; Yan & Han, 2002; Kuramochi & Karypis, 2001; Nijssen & Kok, 2004).

Several different application areas for graph mining are researched. The most common area is mining molecular databases where the molecules are displayed by their two-dimensional structure. When analyzing molecules it is interesting to find patterns that might explain why a certain set of molecules is useful as a drug against certain diseases (Borgelt & Berthold, 2002). Similar problems occur for protein databases. Here graph data mining can be used to find structural patterns in the primary, secondary and tertiary structure of protein categories (Cook & Holder, 2000).

Another application area are web searches (Cook, Manocha, & Holder, 2003). Existing search engines use linear feature matches. Using graphs as underlying data structure, nodes represent pages, documents or document keywords and edges represent links between them. Posing a query as a graph means a smaller graph has to be embedded in the larger one. The graph modeling the data structure can be mined to find similar clusters.

Quite new is the application of subgraph mining in optimizing code for embedded devices. With the help of so-called procedural abstraction, the size of pre-compiled

binaries can be reduced which is often crucial because of the limited storage capacities of embedded systems. There, subgraph mining helps identifying common structures in the program's control flow graph which can then be combined ("abstracted") into a single procedure (Dreweke et.al., 2007).

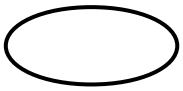

Figure 1 The lattice of all subgraphs in a graph.

BACKGROUND

Theoretically, mining in graph databases can be modeled as the search in the lattice of all possible subgraphs. In Figure 1 a small example is shown based on one graph with six nodes labeled **A,B,C** as shown at the bottom of the figure. All possible subgraphs of this small graph are listed in this figure. At the top of the figure, the empty graph modeled with * is shown. In the next row all possible subgraphs containing just one node (or zeros edges) are listed. The second row contains subgraphs with one edge. The "parent-child" relation between the subgraphs (indicated by lines) is the subgraph property. The empty graph can be embedded in every graph containing one node. The graph containing just one node labeled **A** can be embedded in a one edge graph containing nodes **A** and **C**. Please note, that in Figure 1 no graph with one edge is given containing nodes labeled **A** and **B**. As there is no such subgraph in our running example, the lattice does not contain a graph like this. Only graphs that are real subgraphs are listed in the lattice. In the third row, graphs with two edges are shown and so on. At the bottom of Figure 1, the complete graph with five edges is given. Each subgraph appearing in Figure 1 can be embedded in this graph. All graph mining algorithms have in common, that they search this subgraph lattice. They are interested in finding a subgraph (or several subgraphs) that can be embedded as often as possible in the graph to be mined. In Figure 1 the circled graph can be embedded twice in the running example.

When mining real life graph databases, the situation is of course much more complex. Not only one but a lot of graphs are analyzed leading to a very large lattice. Searching this lattice can be done depth or breadth first. When searching depth first in Figure 1, the first discovered subgraph will be **A** followed by **A-C**, **A-C-C** and so forth.

Thus, first all subgraphs containing **A**, and in the next branch all containing **B** are found. If the lattice is traversed breadth first, all subgraphs in one level of the lattice, i.e. structures that have the same number of edges, are searched before the next level is started. The main disadvantage of breadth first search is the larger memory consumption because in the middle of the lattice a large amount of subgraphs has to be stored. With depth first search only structures which amount is proportional to the size of the biggest graph in the database have to be recorded during the search.

Building this lattice of frequent subgraphs involves two main steps: *Candidate Generation*, where new subgraphs are created out of smaller ones, and *Support Computation* where the frequency or support of the new subgraphs in the database is determined. Both steps are highly complex and thus various algorithms and techniques have been developed to find frequent subgraphs in finite time with reasonable resource consumptions.

MAIN THRUST OF THE CHAPTER

There are two popular ways of creating new subgraphs, new possible candidates for frequent fragments:

1. merging smaller subgraphs that share a common core (Inokuchi et al., 2002; Kuramochi & Karypis, 2004), or
2. extending subgraphs edge by edge (Borgelt & Berthold, 2002; Yan & Han, 2002).

The merge process can be explained by looking at the subgraph lattice shown in Figure 1. The circled subgraph has two parents, **A-C** and **C-C**. Both share the same core

which is **C**. Thus the new fragment **A-C-C** is created by taking the core and adding the two additional edge-node pairs, one from each parent. There are two problems with this approach: First the common core needs to be detected somehow, which can be very expensive. Second a huge amount of subgraphs generated in this way may not even exist in the database. Merging e.g. **A-C** and **B-C** in the example will lead to **A-C-B** which does not occur in the database.

Extending fragments has the advantage that no cores have to be detected. New edge-node pairs (or sometimes only edges, if cycles are closed) are just added to an existing subgraph. In so called *embedding lists,* all embeddings of a fragment into the database graphs are stored. When extending a fragment with an edge (and a node probably), it is easy to check the current embeddings and find edges/nodes connected to these embeddings. This way only existing new fragments are generated. E.g. in Figure 1**,** the embedding list of the circled fragment **A-C-C** contains two embeddings into the only database graph. Checking these embeddings and possible extensions in the database graph leads to new existing fragments.

The importance of a fragment depends on the number of times it appears in the database. If this number is given as a percentage of the number of database graphs, it is called *support.* If an absolute number is given, it is called *frequency*. Two ways of calculating the support/frequency are possible. First the graphs a fragment can be embedded in are counted, no matter how often it can be embedded in one graph. Second the embeddings itself are counted. In Figure 1 the circled fragment appears once in the database in the first case and twice in the second case. Counting embeddings can be done

with subgraph isomorphism tests against all graphs in the database. These tests are NP-complete (Valiente, 2002).

However there is a small improvement for this strategy, as it suffices to check for subgraph isomorphism only in the graphs where the parent graph(s) occur. Unfortunately this requires to keep a list of the graphs in which a subgraph occurs which can be quite memory consuming if the database is large.

The other way of calculating the support is by using the already introduced embeddings lists. An embedding can be thought of as a stored subgraph isomorphism i.e. a map from the nodes and edges in the subgraph to the corresponding nodes and edges in the graph. Now, if the support of a new extended subgraph has to be determined the position in the graph where it can occur is already known and only the additional node and edge have to be checked. This reduces the time to find the isomorphism but comes with the drawback of enormous memory requirements as all embeddings of a subgraph in the database have to be stored which can be millions for small subgraphs on even medium-sized databases of about 25,000 items. Using embedding lists the actual support for a structure can be determined by counting the number of different graphs that are referred to by the embeddings. This can be done in linear time.

Each possible fragment should only be created once, but it is obvious from the lattice in Figure 1, that there may exist several paths through the lattice to reach one fragment. These duplicates must be filtered out. For real life databases, it is also usually not possible to traverse the complete lattice because the number of subgraphs is too large to be handled efficiently. A mechanism is needed to prune the search tree that is built during the discovery process. If the support is calculated based on the number of graphs

a fragment appears in, a supergraph of a graph that is infrequent must be infrequent, too. It cannot occur in more graphs than its parents in the lattice. This property is also known as the *antimonocity constraint*. Once the search reaches a point where a graph does not occur in enough items of the database any more this branch can be pruned. This leads to a drastic reduction of the number of subgraphs to be searched. When the support is calculated based on the embeddings, the antimonocity constraint holds for edge-disjoint embeddings. Therefore after each extension step, the biggest set of edge disjoint embeddings must be calculated. This problem equals the *Maximal Independent Set* problem for graphs, which is also NP-complete (Kuramochi & Karypis, 2005). An independent set is a set of vertices in a graph no two of which are adjacent.

To speed up the search even further various authors have proposed additional pruning strategies that shrink the search tree more efficiently while still finding all frequent fragments. Quite a few algorithms rely on so-called *canonical codes* that uniquely identify a subgraph. A subgraph's code is automatically built during the extension or merging process by e.g. recording the edges in the order as they were added to the graph. After each extension step, this code is compared to the canonical code for this subgraph, which is either the (lexicographically) smallest or biggest possible code. If the subgraph's code is not the canonical one, this branch in the search tree can be pruned, because the same subgraph will be found (or has already been found) in another branch (Borgelt, 2006).

FUTURE TRENDS

Despite the efforts of the last years, still several problems have to be solved. Memory and runtime are a challenge for most of the algorithms. Having real world graph databases containing millions of different graphs, various new algorithms and extensions of the existing ones are necessary. First thoughts concerning this topic can be found in (Wang, Wang, Pei, Zhu & Shi, 2004). Another promising research direction are parallel and distributed algorithms. Distributing the graphs and their subgraph lattice onto different machines or using supercomputers with many processors and much memory can help in processing even larger databases than with current algorithms (Di Fatta & Berthold, 2006; Reinhard & Karypis, 2007).

In several application areas, it is not exact graph matching that is necessary. For example when mining molecules, it is helpful to search for groups of molecules having the same effect but not the same underlying graph. Well-known examples are the number of carbon atoms in chains or several carbon atoms in rings that have been replaced by nitrogen for example (Hofer, Borgelt & Berthold, 2003). In other areas, constraints must be imposed on the fragments to be found, not every possible fragment is helpful. Constraints can e.g. be the density of the graph or the minimal degree of nodes (Zhu, Yan, Han & Yu, 2007).

Additionally visualization of the search and the results is difficult. A semi-automatic search can be helpful. A human expert decides whether the search in a subpart of the lattice is useful or if the search is more promising in another direction. To achieve

this goal a visualization component is necessary that allows browsing in the graph database showing the embeddings of subgraphs.

Finally, another problem of most subgraph mining algorithms is the huge amount of frequent fragments they output. The user has then to scan them all and decide which ones are interesting for the specific application area. Such decision should be incorporated directly into the search process, making the result more understandable and often also the search faster.

CONCLUSION

Graph Data Mining is a currently very active research field. At the main data mining conferences of the ACM or the IEEE every year various new approaches appear. The application areas of graph data mining are widespread ranging from biology and chemistry to compiler construction. Wherever graphs are used to model data, data mining in graph data bases is useful.

REFERENCES

Borgelt, C. & Berthold, M. (2002, December). Mining Molecular Fragments: Finding Relevant Substructures of Molecules. *IEEE International Conference on Data Mining*, ICDM2002. Maebashi City, Japan, IEEE Press, 51-58.

Cook, D. J., & Holder L.B. (2000). Graph-Based Data Mining. *IEEE Intelligent Systems*, 15 (2), 32-41.

Cook, D.J., Manocha, N. & Holder, L.B. (2003). Using a Graph-Based Data Mining System to Perform Web Search. *International J. of Pattern Recognition and Artificial Intelligence*, 17(5), 705-720.

Di Fatta, G. & Berthold, M.R. Dynamic Load Balancing for the Distributed Mining of Molecular Structures. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on High Performance Computational Biology*, vol. 17, no. 8, IEEE Press, 773-785.

Dreweke, Alexander; Wörlein, Marc; Fischer, Ingrid; Schell, Dominic; Meinl, Thorsten; Philippsen, Michael (2007). Graph-Based Procedural Abstraction. *Proceedings of the Fifth International Symposium on Code Generation and Optimization*. IEEE Computer Society, 259-270.

Finn, P., Muggleton, S., Page D. & Srinivasan, A. (1998). Pharmacophore Discovery Using the Inductive Logic Programming System PROGOL. *Machine Learning,* 30 (2-3), 241-270.

Helma, C., Kramer, S. & De Raedt, L. (2002, May). The Molecular Feature Miner MolFea. In: Hicks, M., & Kettner, C. (eds.) *Proceedings of the Beilstein-Institut Workshop Molecular Informatics: Confronting Complexity*. Bozen, Italy.

Hofer, H., Borgelt, C. & Berthold, M. (2003). Large Scale Mining of Molecular Fragments with Wildcards. In: Berthold, M., Lenz, H.J., Bradley, E., Kruse, R., & Borgelt, C. (eds.) *Advances in Intelligent Data Analysis V*, Lecture Notes in Computer Science 2810, Springer-Verlag, 380-389.

Huan J., Wang, W. & Prins, J. (2003). Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. *International Conference on Data Mining*, ICDM2003, Melbourne, Florida, USA, 549-552.

Inokuchi, A., Washio, T. & Motoda, H. (2000, September). An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. *4th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD2000, Lyon, France, 13-23.

Inokuchi, A., Washio, T., Nishimura, K. & Motoda, H. (2002). *A Fast Algorithm for Mining Frequent Connected Subgraphs*. IBM Research, Tokyo Research Laboratory.

Inokuchi, A., Washio, T. & Motoda, H. (2003). Complete Mining of Frequent Patterns from Graphs: Mining Graph Data. *Machine Learning*, 50 (3), 321-354.

King, R., Srinivasan, A. & Dehaspe, L. (2001). Warmr: A Data Mining Tool for Chemical Data. *Journal of Computer-Aided Molecular Design*, 15, 173-181.

Kuramochi M. & Karypis G. (2004, September) An Efficient Algorithm for Discovering Frequent Subgraphs. In: *IEEE Transactions on Knowledge and Data Engineering*, 16 (9), 1038-1051, Piscataway, NJ, USA.

Kuramochi, M. & Karypis G (2005). Finding Frequent Patterns in a Large Sparse Graphs. *Data Mining and Knowledge Discovery*, 11(3): 243-271, 2005.

Nijssen, S. & Kok, J. (2004, April) The Gaston Tool for Frequent Subgraph Mining. *Electronic Notes in Theoretical Computer Science*, 127(1), Elsevier, 77-87.

Reinhard, S. & Karypis, G. (2007, March). A Multi-Level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph. *12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. Long Beach, California USA.

Rückert, U. & Kramer, S. (2004, March). Frequent Free Tree Discovery in Graph Data. In: Haddad, H., Omicini A., Wainwright R., & Liebrock, L. (eds.) *ACM Symposium on Applied Computing*, SAC 2004, Nicosia, Cyprus, 564-570.

Valiente, G. (2002). *Algorithms on Trees and Graphs*. Springer-Verlag.

Wang, C., Wang, W., Pei, P., Zhu, Y., & Shi, B. (2004, August). Scalable Mining Large Disk-based Graph Databases. *10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 2004, Seattle, WA, USA, 316-325.

Yan, X., & Han, J. (2002, December). gSpan: Graph-Based Substructure Pattern Mining. *IEEE International Conference on Data Mining*, ICDM 2002, Maebashi City, Japan, 721-724.

Zhu, F., Yan, X., Han, J. & Yu, P. (2007, May*).* **gPrune: A Constraint Pushing Framework for Graph Pattern Mining**, *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, Nanjing, China.

TERMS AND DEFINITIONS

Antimonicity constraint: The antimonicity constraint states, that any supergraph of an infrequent graph must be infrequent itself.

Candidate generation: Creating new subgraphs out of smaller ones; then it is checked how often this new subgraph appears in the analyzed graph data base.

Canonical code: A unique linear representation of a graph that can be used to prune the search tree

Frequent subgraph: a subgraph that occurs in a certain percentage of all graphs in the database.

Graph isomorphism: Two graphs which contain the same number of graph vertices connected in the same way by edges are said to be isomorphic. Determining if two graphs are isomorphic is thought to be neither an NP-complete problem nor a P-problem, although this has not been proved (Valiente, 2000).

Search tree pruning: Cutting of certain branches of the (conceptual) search tree that is built during the mining process; pruning criteria may be the size of the graphs, the support of the graphs, or algorithm-specific constraints.

Subgraph: A graph G' whose vertices and edges form subsets of the vertices and edges of a given graph G. If G' is a subgraph of G, then G is said to be a supergraph of G'.

Subgraph isomorphism: Decision whether a graph *G'* of is isomorphic to a subgraph another graph *G*. This problem is known to be NP-complete.

Support: The number of graphs or embeddings in the analysed database in which a subgraph occurs.