LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-220300

The State of the Art in Graph-Based Pattern Matching

B. Gallagher

March 31, 2006

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

The State of the Art in Graph-Based Pattern Matching

Brian Gallagher
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
March 31, 2006

ABSTRACT

The task of searching for patterns in graph-structured data has applications in such diverse areas as computer vision, biology, electronics, computer aided design, social networks, and intelligence analysis. As such, work on graph-based pattern matching spans a wide range of research communities. Due to variations in graph characteristics and problem requirements, graph-based pattern matching is not a single problem, but a set of related problems. This paper presents a survey of existing work on graph-based pattern matching, describing variations among graph matching problems, general and specific solution approaches, evaluation techniques, and directions for further research. An emphasis is given to techniques that apply to general graphs with semantic characteristics. The survey also discusses techniques for graph mining, an extension of the graph matching problem.

1. INTRODUCTION

Work on pattern matching in graphs spans a diverse range of research communities within and beyond computer science. Relevant research and application areas include databases, computer vision, mathematical graph theory, artificial intelligence, information retrieval, biology, electronics, computer aided design, and knowledge discovery and data mining.

Graph-based pattern matching is not a single problem, but a set of related problems. These range from the NP-complete *subgraph isomorphism* problem, in which matches are based strictly on graph structure, to finding occurrences of complex structural and semantic patterns in semantic graphs with millions of typed and attributed vertices and edges. The focus of this survey is on techniques that are applicable to general graphs that may have semantic characteristics. I do not cover specialized approaches for structural subclasses of graphs, such as trees or planar graphs.

In the remainder of this section I present a formal description of the basic graph pattern matching problem and discuss a number of common problem variations. In section 2, I outline a general strategy that has been applied by many approaches to matching patterns in graphs. In section 3, I discuss a number of specific approaches that have been developed for solving a variety of graph matching and mining problems. Section 4 covers techniques for the evaluation of graph pattern matching algorithms. Finally, section 5 summarizes the findings of this survey and discusses directions for future research in graph pattern matching.

1.1 The Graph Pattern Matching Problem

The basic graph pattern matching problem is to find matches for a specified pattern in a graph database. More formally, we are given:

1. A **data graph** $G = (V, E)$, composed of a set of vertices V and a set of edges E. Each $e \in E$ is a pair (v_1, v_2) where $v_1, v_2 \in V$. The vertices and/or edges of G may be typed and/or attributed.
2. A **pattern graph (or pattern query)** $P = (V_P, E_P)$, which specifies the structural and semantic requirements that a subgraph of G must satisfy in order to match the pattern P.

The problem is to find the set M of subgraphs of G that "match" the pattern P (structurally and/or semantically). A graph $G' = (V', E')$ is a subgraph of G if and only if $V' \subseteq V$ and $E' \subseteq E$. Generally, it is also required that E' contain only edges for which both endpoints are in V' (i.e., that G' is a graph). Some formulations of the problem require that P represent a single connected graph and, therefore, that the

subgraphs in *M* be connected. A graph is connected if there exists some path between every pair of its vertices.

The precise definition of a "match" varies from problem to problem, but this definition is generally based on some combination of (1) isomorphism (i.e., structural matching) or near isomorphism between *P* and its matches in *M* and (2) equality or similarity between the types and attribute values of the vertices and edges in *P* and its matches in *M*.

1.2 Problem Variations

There are a number of variations on the basic pattern matching problem. Variation generally occurs along one or more of the following dimensions.

Graph properties

All graphs share the same basic structural elements, vertices and edges, but other structural and semantic graph properties vary from problem to problem. Some graphs are typed. That is, vertices and/or edges are assigned a type (or label) from predefined sets. For example, if we have a graph representing a movie database, vertex types may include *movie*, *actor*, and *director* and edge types may include *appeared-in*, *sequel-to*, and *directed* (see **Figure 1**). Other graphs are not typed (i.e., all vertices or edges are of a single implicit type). In addition to type labels, some systems allow a set of attributes and their corresponding values to be associated with each vertex and/or edge in the graph. Returning to our movie graph example, a *movie* vertex may have attributes such as *title*, *year*, *genre*, and *rating*.

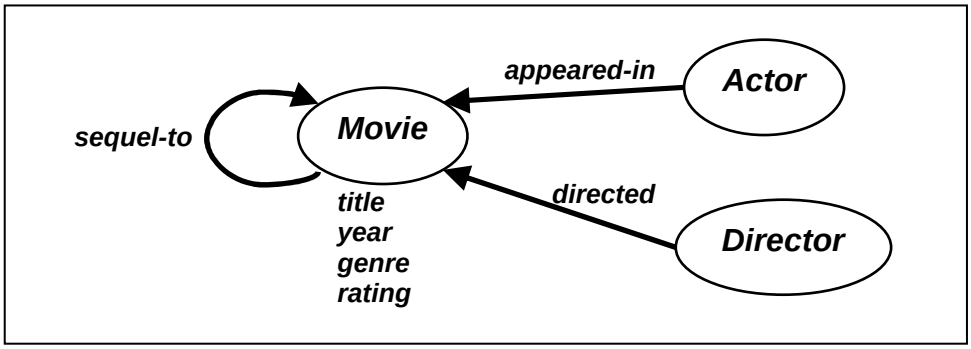

Figure 1: An example ontology for a movie database

A *semantic graph* is a graph-structured data representation in which vertices represent concepts and edges represent relationships between concepts (as in the movie database example). In general, both vertices and edges in a semantic graph are typed and attributed, as discussed above. Furthermore, a semantic graph generally has an associated *ontology*, which specifies the concepts that may appear in the graph, the relationships allowed between each pair of concepts, and the attributes associated with each concept (vertex type) and relationship (edge type). Ontologies themselves are often represented as graphs (see **Figure 1**).

Graphs also differ in the restrictions placed on edges. For example, graphs may allow or disallow directed or undirected edges, weights on edges, multiple edges between a pair of vertices (multigraphs), and edges that connect a single vertex to itself (self-loops).

Single-graph vs. graph-transaction setting

Some graph databases consist of a set of relatively small graphs (called *transactions*), while others consist of a single large graph. Kuramochi and Karypis [22] refer to these cases as the *graph-transaction setting* and the *single-graph setting*, respectively. The single-graph setting is more general and algorithms developed for this setting can be readily applied to the graph-transaction setting, although the converse is generally not true.

Structural vs. semantic matching

Since many graphs are neither typed nor attributed, many graph matching algorithms perform matching based strictly on link structure (i.e., the "shape" of the graph) and completely ignore the semantics (or meaning) associated with the graph's vertices and edges [24,25,31,34]. Other algorithms take semantics into account by matching based on vertex/edge types or attributes as well as structure [1,8,10,17,36].

Exact vs. inexact matching

A graph matching algorithm may return only those results that match a specified pattern exactly or it may return a ranked list of the most similar matches, as is common in an information retrieval setting (e.g., Web search engines). In addition, some systems allow patterns to be left only partially specified (e.g., using "wildcards" or cardinality operators) [4,36]. In this case, while the results may match the pattern exactly, this is a form of inexact matching because the pattern itself is inexact (e.g., "find all *movie* vertices linked to any vertex" vs. "find all *movie* vertices linked to an *actor* vertex").

Exact vs. approximate solutions

Quite distinct from whether an algorithm performs exact or inexact matching, algorithms vary based on their guarantees regarding solution quality. Exact algorithms for *subgraph isomorphism* (i.e., structural pattern matching) are guaranteed to find an optimal solution, but have exponential worst-case complexity [29]. Approximate algorithms [7,32] often have polynomial complexity, but are not guaranteed to find an optimal solution.

Graph matching vs. graph mining

There are a variety of problems that build on graph pattern matching. One such problem, which has received increasing attention recently, is that of *graph mining* or *structural motif finding*. Whereas the goal of graph matching is to find occurrences of a specific pattern in a graph, the goal of graph mining is to find a set of the most common or most "interesting" patterns in a graph [9,11,16,22,34,38].

2. A GENERAL-PURPOSE MATCHING APPROACH

Subgraph isomorphism is the problem of determining whether one graph *P* is isomorphic to a subgraph of another graph *G* (i.e., whether the pattern *P* has a structural match in *G*). Since subgraph isomorphism is known to be NP-complete [34], all known algorithms for finding subgraph isomorphisms are exponential in the size of the input graphs. Therefore, it is impractical to solve subgraph isomorphism directly in large graphs. This leaves us with two options for matching patterns in reasonable time: (1) use an approximate isomorphism algorithm, which may yield non-optimal solutions (i.e., false positives or false negatives) or (2) use an exact isomorphism algorithm, but apply it to only a subset of the data. In general, this second approach is achieved by performing some pre-processing to filter out unpromising portions of a dataset before any direct matching takes place. This data filtering step is known as *candidate selection*.

Giugno and Shasha [15] describe the three basic components of their algorithm, *GraphGrep*, as index construction, database filtering, and subgraph matching. This basic framework can be generalized to describe the majority of graph pattern matching algorithms. In general, these algorithms have three phases: data analysis and metadata construction, candidate selection, and matching.

Data analysis and metadata construction

Many algorithms perform some sort of pre-processing on a dataset to create summary information, which informs the candidate selection process. *Graph invariants* are a common example of such summary information. An invariant is a quantity used to characterize a graph [34]. If two graphs are identical, they will have identical invariants, although the converse is not necessarily true. Due to this property, a simple comparison of invariant values between pattern and data graphs may be sufficient to eliminate many non-matches. Graph invariants are most common in the graph-transaction setting and are generally applied to exact matching. The *nauty* algorithm [24] computes graph invariants for each vertex in a graph (e.g., degree). Several algorithms use a canonical graph representation to derive invariants [34]. GraphGrep [15] creates a "fingerprint" for each graph in a database using path-based invariants.

A complimentary approach to calculating graph invariants is to create a statistical summary of an entire graph dataset. For example, Statistical Relational Models (SRMs) [13] construct a model of the dependencies between attributes in a relational dataset by utilizing conditional independence properties between attribute values within and across relations (i.e., database tables). SRMs can provide approximate answers to counting queries (e.g., how many *movies* in our dataset have *actors* with the *last name* "Smith"?). These approximate counts are helpful for determining an efficient ordering of filtering criteria when multiple criteria exist for eliminating non-matches (e.g., multiple graph invariants).

In addition to the types of summary information already mentioned, it is often useful to construct one or more indices into a graph dataset, as is common in relational databases. Indices into graph data can take many forms. For example, we may want to index occurrences of common structural patterns in a large graph so that we can locate them quickly. The *embedding lists* used by graph mining algorithms are a common example of this (see section 3.2). In semantic graphs, it is common to retrieve all vertices or edges of a particular type or with a particular attribute value (e.g., 'find all *movies*' or 'find all *actors* with the last name "Smith"'). These operations can be sped up using indices on types and attributes.

Candidate selection

Once we have constructed metadata (e.g., an index, measure, or model) for our dataset, we can use this information to direct our search for matching subgraphs. If graph invariants differ between a subgraph *S* and our pattern *P*, there is no need to perform direct matching between *P* and *S*. If an SRM tells us that a match to our query is very unlikely, we may not bother searching for an exact match. Statistical models such as SRMS can also help us decide which non-matches to filter out first by providing *selectivity estimates* (i.e., determining which criteria in a pattern are the most selective or occur least frequently in the dataset). The TRAKS [1] and LAW [36] systems use simple frequency statistics to perform this kind of selective pruning on the space of potential matches.

It is worth noting that effective candidate selection in semantic graphs is possible without the explicit generation of graph invariants since semantic graphs already contain a rich set of data on which to filter potential matches (i.e., the types and attribute values attached to graph vertices and edges). For example, if our pattern contains a vertex *M* of type *movie*, we don't need to consider every vertex in a data graph as a potential match to *M*, only those vertices of type *movie*. Even so, type and/or attribute based indices into our dataset may speed up this filtering process and statistical summaries, such as those provided by SRMs, may prove useful for creating an efficient ordering of filters.

When the candidate selection phase is complete, we have a list of candidates on which to perform matching. The goal of candidate selection is to produce as small a list of candidates as possible without eliminating any true matches from the list.

Matching

During the matching phase, candidates identified by candidate selection are checked for matches to the specified pattern. Researchers have experimented with a number of approaches for solving graph matching problems. The next section provides details on many specific approaches.

3. SPECIFIC MATCHING AND MINING APPROACHES

Researchers have explored a number of approaches for pattern matching and mining in graphs. These include [34]:

- Search-based techniques
- Inductive Logic Programming
- Inductive databases
- Mathematical graph theory (such as Complete Level-Wise Search)
- Kernel functions (such as Support Vector Machines)

The focus of this survey is on search-based solutions.

The remainder of this section is organized as follows. First I discuss matching and mining approaches that perform matching based on graph structure (i.e., subgraph isomorphism). Then I present several matching techniques that make use of semantics as well as structure. Finally, I describe approaches for measuring similarity between graphs to support inexact pattern matching.

The set of approaches covered here is illustrative, but not exhaustive. For a more complete listing of graph matching approaches see Shasha et al. [29]. For a more complete listing of graph mining approaches, see Washio and Motoda [34] and Wörlein et al. [38].

3.1 Structural Matching Approaches

One of the most highly-cited approaches to exact pattern matching is the subgraph isomorphism algorithm proposed by Ullmann [31]. This algorithm operates on single untyped graphs with directed or undirected edges. Suppose we want to find matches to the pattern graph *P* in the data graph *G* depicted in **Figure 2**.

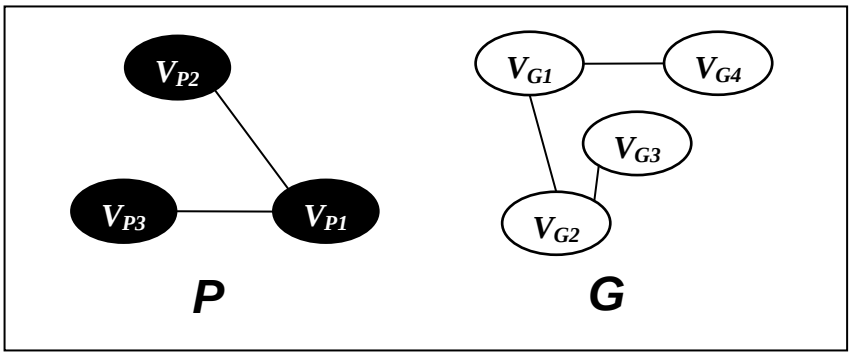

Figure 2: An example pattern graph *P* and data graph *G*

The basic approach is to enumerate all possible mappings of vertices in *P* to those in *G* using a depth-first tree-search algorithm. Each node at level *i* of the search-tree maps vertex V_{Pi} in *P* to a different vertex in *G* (**Figure 3**). Each path from root to leaf in the search-tree represents a complete mapping of the vertices in *P* to those in *G*. Any such mapping that preserves adjacency in *P* and *G* (i.e., vertices that are neighbors in *P* map to vertices that are neighbors in *G*) represents an isomorphism from *P* to a subgraph of *G*. If no such mapping preserves adjacency, then no such isomorphism exists. Since the search-space considered by this approach increases exponentially with the size of the input graphs, Ullmann suggests a refinement procedure to prune unpromising sub-trees, eliminating the need to expand them. This procedure eliminates vertex mappings from consideration based on three criteria:

1. **Vertex degree** – If the degree of vertex V_{Pi} (i.e., the number of edges adjacent to V_{Pi}) is greater than the degree of V_{Gj} then V_{Pi} cannot possibly map to V_{Gj}. For example, since V_{P1} has degree two and V_{G4} has degree one, no match can map V_{P1} to V_{G4} (**Figure 2**).
2. **One-to-one mapping of vertices** – Once we decide to map V_{Pi} to V_{Gj}, along a particular path through the tree, we cannot map to V_{Pi} any other vertex in *G* and we cannot map any other vertex in *P* to V_{Gj}.
3. **Forward checking** – As we work our way down the tree, for any possible vertex mapping that remains, we can eliminate the mapping if it cannot possibly preserve adjacency between *P* and *G*. For example, suppose that we have decided to map V_{P1} to V_{G1} and we are considering the possible mapping from V_{P2} to V_{G3}. Regardless of what we do further down the tree, mapping from V_{P2} to V_{G3} cannot possibly preserve adjacency since V_{P1} and V_{P2} are neighbors in *P*, but V_{G1} and V_{G3} are not neighbors in *G*. So, we can eliminate the possible mapping from V_{P2} to V_{G3} from further consideration.

As Ullmann's algorithm expands a particular path in the search-tree, one of two things will happen:

1. The algorithm will eliminate all possible mappings for some vertex in *P*. In this case, the path we are on cannot yield a match. We can safely stop, without expanding additional nodes along the current path, and backtrack.
2. The algorithm will reach a leaf of the tree, having successfully mapped each vertex in *P* to exactly one vertex in *G*. In this case, the path represents a match for *P* in *G* (see **Figure 3**).

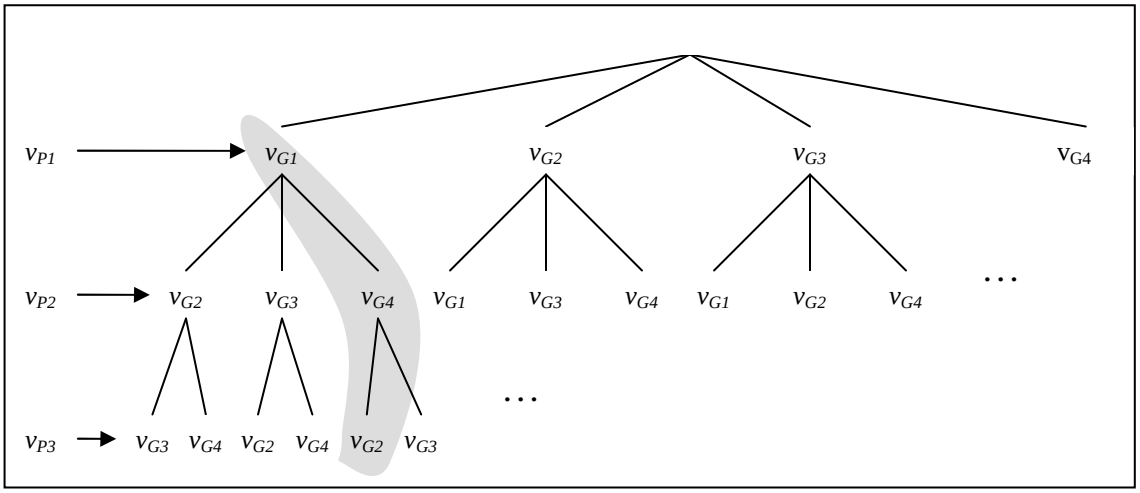

Figure 3: A partial search-tree for Ullmann's algorithm, mapping the vertices v_{P1}, v_{P2}, v_{P3} in pattern graph *P* to vertices v_{G1}, v_{G2}, v_{G3}, v_{G4} in graph database *G*. The highlighted path represents a match for *P* in *G*.

Messmer and Bunke [25] point out that, despite the refinement procedure, Ullmann's algorithm has worst-case time-complexity that is exponential in the size of the input graphs. They propose an alternative method for exact subgraph isomorphism that has only quadratic worst-case time complexity. Their algorithm also operates on multiple untyped graphs with directed or undirected edges. The approach is to pre-process the graph dataset to generate all possible permutations of the graph adjacency matrices offline and organize them in a decision tree. At run time the decision tree is used to classify the adjacency matrix of the pattern graph. The drawback to this approach is that the size of the decision tree grows exponentially with respect to the size of the data graph. To address this issue, the authors present pruning techniques, which are effective in reducing decision tree size. However, the pruned decision trees can no longer guarantee polynomial run times.

The *nauty* algorithm [24,34] detects isomorphism between untyped graphs that may be directed or undirected. *Nauty* uses transformations to reduce graphs to a canonical form that may be checked relatively quickly for isomorphism. Specifically, the algorithm computes invariants for each vertex in a graph (e.g., degree and counts of adjacent vertices of various degrees) that are used for candidate selection. *Nauty* partitions a graph into non-overlapping subsets such that the vertices in a particular subset share identical invariant values. Subsets having the same invariant values can then be compared across graphs. If all subsets are isomorphic between two graphs, then the two graphs must be isomorphic. Alternatively, if two graphs contain subsets with differing invariants, there is no need to test isomorphism between the sets directly.

SUBDUE [9,11] operates in a single-graph setting with typed vertices and typed, directed edges. SUBDUE is a graph mining system, but performs pattern matching as an intermediate step in the mining process. The algorithm uses a graph matching approach based on that of Bunke and Allermann [6]. The basic approach is similar to Ullmann's algorithm. We construct a search-tree, where the nodes at the i^{th} level map the i^{th} vertex from *P* to some vertex in *G*. A path through the tree represents a complete mapping of vertices from

P to *G*. Since SUBDUE performs inexact matching, each node in the search-tree also has an associated cost that captures how well *P* matches *G*. The cost is a distance measure between graphs. If *P* and *G* are exactly isomorphic, there will be a mapping between them with 0 cost. The more different *P* and *G* are, the higher the cost will be. Match costs are calculated as the sum of the costs assigned to the individual "distortions" required to convert *P* to *G*. Distortions are described in terms of basic transformations such as deletion, insertion, and substitution of vertices and edges. For the inexact matching task, the goal state is the final state (leaf) with the least cost of all final states. Since the search-space is again very large, SUBDUE applies a branch-and-bound search to the tree. Because branch-and-bound is guaranteed to find an optimal solution, the search terminates once any complete mapping is found. The algorithm also allows an upper limit to be placed on the number of search nodes considered, which can lead to a significant savings in search time at the expense of solution quality.

3.2 Structural Mining Approaches

Graph mining algorithms find "interesting" patterns in graphs, based on some definition of interestingness. For instance, the SUBDUE system evaluates the interestingness of patterns based on the amount of information required to represent them and the frequency of their occurrence. More commonly, the patterns of interest are simply those that occur most frequently in the graph database. Interesting patterns are generally found by enumerating possible patterns (i.e., *candidate generation*) and then evaluating each pattern in terms of its interestingness. In cases where it is the frequency of patterns that is interesting, evaluation consists of checking generated candidate patterns against a graph database and counting how frequently each pattern occurs. Wörlein et al. [38] describe three main problems that all efficient subgraph mining systems must solve:

1. **Purposive refinement** – Efficiency of mining algorithms improves if an algorithm enumerates only those patterns that actually appear in the database instead of exhaustively enumerating all possible patterns.
2. **Efficient enumeration** – Mining algorithms must be able to detect and prune duplicate patterns in order to avoid counting duplicates and performing unnecessary processing. Since duplicate detection generally requires an isomorphism test, it is desirable to prevent generation of duplicate patterns in the first place. In addition, comparing patterns using a canonical graph representation can improve duplicate detection times over performing explicit isomorphism calculations.
3. **Focused isomorphism testing** – Some graph mining systems store *embeddings* (or *embedding lists*), which map the vertices and edges from a pattern into the data graph(s) where the pattern occurs, allowing quick access to these occurrences. If larger candidate patterns are generated by extending smaller patterns, checks for the larger pattern need only to be carried out in portions of the data graph(s) where the smaller pattern is known to occur. Thus, embeddings can reduce the number of subgraph isomorphism tests required.

SUBDUE [9,11] performs substructure discovery (i.e., graph mining) in a single-graph setting with typed vertices and typed, directed edges. SUBDUE uses minimum description length (MDL) to discover common graph substructures. Specifically, SUBDUE searches for the substructure *S* that minimizes the number of bits required to store *S* plus the original graph with *S* replaced by a single vertex. The goals are (1) to find the substructures that allow the entire graph to be maximally compressed and (2) to identify conceptually interesting substructures. Substructures are discovered using a computationally-constrained beam search. Like best-first search (e.g., A*), beam search uses a heuristic function to decide which search nodes to expand. However, beam search only stores the *k* most promising nodes at any time (*k* is referred to as the *beam width*), so the algorithm is approximate and not guaranteed to find an optimal solution. The algorithm begins with substructures that match a single vertex in the data graph and then expands the substructures by one edge on successive iterations of the algorithm. In this way the algorithm considers only those patterns that actually appear in the data (i.e., purposive refinement). The algorithm searches for the best structure (based on MDL) until all substructures have been exhausted or the algorithm exceeds specified computational constraints. The algorithm is capable of discovering more complex substructures by making successive passes over the data, producing a hierarchy of structural "concepts." SUBDUE uses either exact or inexact matching of structures. Details of the graph matching algorithm are discussed above

in Section 3.1. It is unclear from the algorithm description whether SUBDUE uses focused isomorphism testing or how SUBDUE addresses the efficient enumeration problem.

Kuramochi and Karypis [22] also address the problem of graph mining in a single-graph setting. Their graphs are typed and undirected. They offer solutions to the "exact discovery problem," but also faster solutions to the "approximate discovery problem" (a subset of the exact problem) and the "upper bound discovery problem" (a superset of the exact problem). Their general approach is to construct a lattice of frequent subgraphs. Level *k* in the lattice contains all frequent subgraphs of size *k* (i.e., *k* edges). The authors present two algorithms for generating the lattice and the associated frequencies of nodes (representing subgraphs) in the lattice. The HSIGRAM algorithm generates the lattice breath-first and the VSIGRAM algorithm generates the lattice depth-first. Both algorithms employ a number of optimizations to speed up processing. In particular, since solving subgraph isomorphism is a frequent operation, they keep around smaller frequent structures and then build on them to create larger structures (i.e., subgraphs further down in the lattice). This technique for selective candidate generation is similar to the idea of candidate selection for matching and addresses the purposive refinement problem mentioned above. VSIGRAM implements focused isomorphism testing by storing complete embeddings and HSIGRAM stores incomplete but compact embedding information in the form of *anchor edges*. Kuramochi and Karypis also address the efficient enumeration problem. For example, HSIGRAM reduces the number of duplicate candidates generated by only joining previously generated frequent subgraphs of size *k* that share a certain "properly selected" subgraph of size (*k*-1).

Goethals et al. [16] propose an algorithm for mining frequent tree-structured patterns in a single untyped, directed graph. In addition to the usual pattern vertices that may map to any vertex in the data graph (with matching structure), their algorithm finds patterns that contain *selected* vertices and *existential* vertices. Selected vertices are labeled by constants and must map to specific vertices in the data graph that are identified by those constants. Existential vertices are used like any other vertex to determine matches, but are not used in counting pattern occurrences. The algorithm incrementally generates all possible tree structures, starting with the smallest and increasing in size up to some limit. The algorithm takes advantage of properties of trees to efficiently avoid the generation of duplicates (i.e., efficient enumeration). For each tree *T*, the algorithm generates all possible patterns based on *T* and executes a SQL query to check for matches to the pattern. Duplicate patterns are detected efficiently and removed using a canonical pattern representation. Goethals et al. address the purposive refinement problem by generating candidate patterns for a particular *T* starting with the most general and only generating further candidates whose generalizations are already known to be frequent. Their procedure does not use focused isomorphism testing, but employs optimizations that allow the frequencies for sets of related patterns to be computed in parallel.

Many recently proposed algorithms (e.g., MoFa [5], gSpan [39], FFSM [20], Gaston [26]) are similar to the approaches described here. All of these recent approaches generate candidate patterns in a depth-first rather than breadth-first fashion [38].

3.3 Semantic Matching Approaches

So far, I have discussed techniques that match patterns based on graph structure. Here, I present techniques that make some use of graph semantics (i.e., vertex and edge types and attribute values) as well as structure.

As previously discussed, SUBDUE determines similarity between graphs using a cost function that is based on the distortions necessary to transform one graph into another. These transformations may be purely structural or they may be based on graph semantics as well. Djoko et al. [11] explain that transformation costs in SUBDUE may also be based on domain knowledge (i.e., on the underlying ontology). They suggest, for example, that the cost of a vertex substitution could vary based on the vertex types involved.

GraphGrep [15] operates in the graph-transaction setting on undirected graphs with typed vertices. The algorithm makes implicit use of vertex type information to perform matching. Matching in GraphGrep relies on the concept of a label path, which is simply a sequence of type labels along a path in a graph (e.g., actor-movie-director-movie-actor). As mentioned above, the GraphGrep algorithm consists of three basic components: index construction, database filtering, and subgraph matching. During index construction, the algorithm computes a "fingerprint" for each graph in the database. The fingerprint of a graph is essentially a

set of pairs ⟨*h*(*labelPath*), *numIDpaths*⟩, one for each label path in the graph. Here *h* is a hash function and *numIDpaths* is the number of instances in the graph of the specified label path. During candidate selection, the database is filtered based on the fingerprint of the query. Specifically, if a graph *G* has a lower *numIDpaths* value than the query for any *labelPath* in its fingerprint, then it cannot contain an exact match for the query and can be removed from consideration. During the subgraph matching phase, the query is broken up into sequences of overlapping label paths, which are compared against the candidate graphs. Label paths of the candidate graphs that match the query label paths can then be combined into a single matching subgraph.

TMODS [8,10,17] uses a set of genetic algorithms to find exact and inexact pattern matches in directed, attributed graphs. Patterns may specify both structural and attribute characteristics. TMODS searches for patterns from the bottom-up, finding sub-patterns first and then composing them into more complex higher-level patterns. The TMODS pattern matching algorithm is not described in further detail.

TRAKS [1] performs inexact pattern matching in typed, directed graphs on vertex and edge types as well as structure. Matches are ranked by similarity to the original pattern, taking into account ontological distance between types. Entities in a pattern are processed in ascending order of the frequency of their type in the dataset, to eliminate non-matches more quickly. The algorithm searches for matches in a depth-first fashion by expanding partial matches by one vertex or edge at a time.

The LAW system [36] performs inexact pattern matching on typed, directed graphs. Patterns are represented as graphs with typed vertices and edges. In addition, the pattern language supports construction of more sophisticated pattern queries through constraints between vertices and notions such as hierarchy (i.e., sub-patterns), disjunction, and cardinality (i.e., the number occurrences of a vertex or edge). LAW uses *graph edit distance* to measure similarity between potential matches. Graph edit distance is defined as the minimum number or cost of edit operations required to transform one graph into another. Edit operations include deletion and replacement of vertices and edges. LAW uses ontological distance to measure differences between types. Like many of the matching techniques already discussed, LAW finds matches using a search-tree. The LAW search algorithm is based on A* [19] and selects nodes for expansion based on the minimum worst-case cost. This cost is calculated based on the true cost of the mappings so far plus the cost of deleting all unexplored vertices and edges in the pattern. Although this heuristic is not admissible (in fact, it is an upper bound on the actual cost), LAW is guaranteed to find the lowest-cost matches because, unlike pure A*, the LAW algorithm uses the worst-case cost heuristic only as a selection rule and not as its termination condition [35].

LAW generates start states by selecting the vertex in the pattern with the fewest legal mappings in the data and partial matches are expanded by selecting unexplored vertex mappings and generating adjacent edge mappings. LAW uses an "anytime" version of A* that may be terminated at any point and will return the matches it has found so far. The set of matches is guaranteed to monotonically improve as the algorithm continues to run. The LAW pattern matcher uses a "search plan" (i.e., query execution plan) to determine the order in which query elements are processed. Search plans may be specified by the user or automatically calculated based on a "statistical analysis of the data." The authors do not describe the details of this statistical analysis [37].

Statistical Relational Models (SRMs) model the joint distribution over tuples in a relational database and capture the frequencies with which the tuples join [13]. Although this work does not provide an explicit pattern matching algorithm, it demonstrates that SRMs exhibit substantially lower relative error than previous methods (i.e., methods that assume attribute independence or join uniformity) for estimating the size of a query's result set (i.e., selectivity estimation). This suggests an approach for optimizing pattern queries for semantic graphs using selectivity estimation and query optimization techniques, as have been studied in the XML and database communities [3,14,18,21,27,33].

3.4 Similarity-Based Matching Approaches

Inexact matching approaches, such as SUBDUE, TMODS, TRAKS, and LAW (all discussed above), match and rank results based on their "similarity" to a specified pattern. Such approaches require a well-defined similarity measure in order to compare graphs.

As described above, SUBDUE and LAW each use variations on the idea of graph edit distance to determine similarity between graphs. Graph edit distance is defined as the minimum number or cost of edit operations required to transform one graph into another. Edit operations typically include deletion, insertion, and replacement of vertices and edges. Graph edit distance generally measures only the structural similarity between graphs. Since graph patterns may specify type and attribute characteristics as well, a pattern matching algorithm for semantic graphs may also need to take similarity between attribute values and vertex and edge types into account.

Attributes may be compared using any number of similarity measures. Many attribute similarity measures are data-type dependent (e.g., Euclidean distance, string edit distance, cosine similarity). Other similarity measures are more general. For example, Lin [23] presents an information-theoretic definition of similarity and shows how it may be applied to strings, feature vectors, ordinal values, words, and concepts in a taxonomy. With the exception of TMODS, none of the pattern matching algorithms surveyed here appear to match based on attribute values. The TMODS literature does not provide details on attribute matching.

When graph edit distance is used in the context of semantic graphs, the measure may be extended to capture the semantic similarity between types in addition to structural similarity. When the edit operation of replacing a vertex or edge is included, a cost may be assigned to this operation based on the ontological distance between the types involved. SUBDUE, TRAKS, and LAW are each apparently able to take ontological distance into account when matching, although the exact method of determining ontological distance in each is unclear.

Many proposed ontological distance measures are based on the length of the path between concepts (i.e., types) in a concept hierarchy (e.g., Rada et al. [28]). Other measures take into account both the concept hierarchy and the way that concepts are used in the dataset. For example, by Lin's information-theoretic definition, similarity between concepts is measured as the ratio between the amount of information needed to state the commonality of the concepts and the amount information needed to fully state each of the concepts. The basic intuition here is that if two concepts are similar, their commonality will be larger and their differences will be smaller. Applying a standard information-theoretic definition of information content, Lin's similarity definition amounts to: (1) concepts *A* and *B* are more similar the less their nearest common ancestor occurs in the data (i.e., the more specific the common ancestor) and (2) *A* and *B* are more similar the more *A* and *B* occur in the data (i.e., the less specific they are). For example, if you know that *A* and *B* are both specializations of the concept *thing*, you won't have a particularly strong belief that *A* and *B* are similar. However, if you then learn that *A* and *B* are both specializations of *food* (which is more specific and less common than *thing*), your belief in the similarity of *A* and *B* will increase. Now suppose you know that *A* is a *fruit* and *B* is a *vegetable*. At this point you may have a fairly strong belief that *A* and *B* are similar. If you then learn that A is a *watermelon* and B is *spinach*, this is likely to decrease your belief that *A* and *B* are similar.

4. EVALUATION

As might be expected given the number of research communities involved in work on graph matching over the past several decades, it is difficult to evaluate the performance of the various techniques in relation to one another. Different algorithms have differing goals and researchers have evaluated their algorithms on datasets that vary tremendously in terms of size and graph characteristics. In addition, the complexity of the patterns evaluated is a huge potential source of variation among the results of various studies. Evaluations generally have not attempted to analyze or quantify the complexity of the patterns used for evaluation.

Work on graph pattern matching to date has focused evaluation on the runtime performance of algorithms. Even for the inexact matching techniques I surveyed, there was no systematic evaluation of solution quality (e.g., precision and recall of matches). The most common evaluation metric for the algorithms surveyed here is runtime vs. dataset size. In the graph-transaction setting dataset size generally means the number of graphs in the dataset. In the single-graph setting, dataset size is generally the number of vertices in the graph. There are a number of variations. For example, Ullmann [31] reports runtime vs. the number of matches in the graph (i.e., selectivity of the pattern query), Wolverton et al. [30] (LAW) report runtime vs. the number of edges in the graph, and Messmer and Bunke [25] report runtime, computation steps, and

decision tree size vs. the number of vertices and edges in the graph, the number of graphs, and the decision tree depth.

The datasets used for evaluation by most systems mentioned in this survey are synthetically generated graphs with either random or regular linkage (e.g., mesh structure) between vertices. These graphs also tend to be quite small. For example, Ullmann's graphs have between 12-14 vertices and ~20-30 edges, the graphs used by Messmer and Bunke have up to 29 vertices and up to 44 edges, and *nauty* uses graphs with between 10-1000 vertices (the number of edges is not given). The graphs used to evaluate LAW are the largest, with between 12,000 and 240,000 edges (the number of vertices is not given). GraphGrep [15] was evaluated on National Cancer Institute databases containing up to 16,000 individual graphs representing molecules. These graphs contain an average of 20 vertices and a maximum of 270 (the number of edges is not given).

In addition to the evaluations carried out in individual studies, researchers have recently begun benchmarking activities, comparing various algorithms for graph matching and mining. So far, these comparisons have also been based strictly on the runtime performance of algorithms. The more difficult problem of comparing algorithms based on their output (e.g., accuracy, precision/recall, or utility measures for inexact matching and mining) has largely been ignored.

Foggia et al. [12] compare five graph isomorphism algorithms (including Ullmann and *nauty*, described above) in the graph-transaction setting on four kinds of untyped, synthetic graphs: random, regular 2D mesh, irregular 2D mesh, and bounded valence (i.e., bounded degree). The authors present the results of their benchmarking experiments, but provide little interpretation. The main conclusion to draw is that graph isomorphism algorithms appear to be sensitive to the structure of the input data, so it is important to consider graph characteristics when choosing or developing an algorithm. This also suggests that much of the existing evaluation may not tell us much about the applicability of algorithms to large real-world datasets.

Wörlein et al. [38] implement and compare four subgraph mining algorithms in the graph-transaction setting on general, undirected graphs with typed vertices and edges. Runtime comparisons are performed on several real molecular databases and synthetic datasets. This study breaks down the evaluation into several specific subproblems: duplicate filtering/pruning, pattern counting, embedding list calculations, extending of subgraphs, and joining of subgraphs. The authors make a number of general conclusions based on their analysis:

- Embedding lists do not considerably speed up search in general. Their utility increases as the mined subgraphs become larger.
- Candidate generation, pattern counting, and embedding list computations are much more important to performance than strategies for pruning duplicate patterns.
- Duplicate detection using a canonical graph representation is more efficient than explicit isomorphism testing.
- All algorithms tested scale linearly with database size (i.e., the number of graphs per database), though with different factors.

5. SUMMARY

The problem of searching for patterns in graph-structured datasets has applications in such diverse areas as computer vision, biology, electronics, computer aided design, social networks, and intelligence analysis. Accordingly, numerous techniques have been developed for matching and mining patterns in graphs. Together, these techniques represent decades of work by researchers from a diverse range of research communities. Despite the variations in properties of graphs, databases, and algorithms, several common themes have emerged. Since subgraph isomorphism algorithms are computationally expensive, keeping isomorphism calculations to a minimum is crucial to algorithm performance. Candidate selection is an effective means by which to accomplish this and techniques for metadata construction and application (e.g., indexing and data summarization, compression, and modeling) are central to effective candidate selection.

While there has been some work on pattern matching using semantics as well as structure, there appear to be opportunities to further exploit graph semantics for indexing, candidate selection, and matching. Based on the work reviewed in this survey, I have made the following observations about pattern matching in semantic graphs:

- Existing tree-search techniques (e.g., Ullmann's algorithm) can be readily extended to match and prune based on semantics as well as structure.
- Existing evaluation focuses on relatively small graphs, often with connections that are either random or regular (e.g., a mesh or bounded degree vertices). Real-world semantic graphs (e.g., social networks and the World Wide Web) are large and exhibit scale-free network characteristics (e.g., power-law degree distribution, high clustering coefficient, and short characteristic path length) [2]. Therefore, existing evaluations tell us little about the applicability of algorithms to matching in semantic graphs.
- Many existing matching algorithms focus on the graph-transaction setting, where individual graphs tend to be very small. Therefore, many techniques are not directly applicable to large graphs (e.g., millions of vertices). Note that if candidate selection is effective in breaking large graphs into smaller graphs, techniques developed for the graph-transaction setting may be successfully applied to these smaller graphs.
- Existing candidate selection and indexing strategies focus on graph structure. Type and attribute information can potentially help filter out irrelevant data more quickly. However, more sophisticated graph statistics are required to capture the combination of attributes, type, and structure.
- Existing graph similarity measures do not incorporate all of attributes, type, and structure. An important question for inexact matching in semantic graphs is how to combine these different kinds of similarity. For example, if G_1 and G_2 are structurally similar, but have different attribute values and G_1 and G_3 have similar attribute values, but differ structurally, which of G_2 and G_3 is a better match for G_1?

Based on these observations, the following appear to be promising research directions for pattern matching in semantic graphs:

- Application of query optimization techniques from relational and XML databases to graph databases. Specifically, the optimization of pattern queries based on selectivity estimates derived from probabilistic relational models. This may include learned models such as SRMs as well as statistical, mathematical, or probabilistic models that do not require learning, but are based on simple measures calculated from the data graph.
- Techniques for indexing and candidate selection that utilize both graph structure and semantics (i.e., vertex and edge types and attribute values).
- Techniques for inexact matching that utilize both graph structure and semantics (i.e., distance measures that incorporate ontological distance between types as well as differences in attribute values and graph structure).
- Evaluation of matching algorithms on large real-world semantic graph datasets. Generation of synthetic datasets that reproduce characteristics of real-world data. Evaluation techniques that take into account the complexity and selectivity of the patterns used for evaluation.

6. ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48. UCRL-TR-220300. Many thanks to Tina Eliassi-Rad for helpful comments on this survey at various stages.

7. REFERENCES

[1] B. Aleman-Meza, C. Halaschek-Wiener, S. S. Sahoo, A. Sheth, and I. B. Arpinar, "Template Based Semantic Similarity for Security Applications," *IEEE International Conference on Intelligence and Security Informatics (IEE ISI-2005)*, 2005.

[2] A. Barabási and A. Reka, "Emergence of Scaling in Random Networks," *Science*, 286:509-512, October 15, 1999.

[3] L. Becker and R. H. Guting, "The GraphDB Algebra: Specification of Advanced Data Models with Second-Order Signature," Informatik-Report 183, FernUniversitat Hagen, Praktische Informatik IV, 1995.

[4] H. Blau, N. Immerman, and D. Jensen, "A visual language for querying and updating graphs," University of Massachusetts Amherst Computer Science Technical Report 2002-037, 2002.

[5] C. Borgelt and M. R. Berthold, "Mining Molecular Fragments: Finding Relevant Substructures of Molecules," *Proceedings IEEE International Conference on Data Mining (ICDM)*, 2002.

[6] H. Bunke and G. Allermann, "Inexact Graph Matching for Structural Pattern Recognition," *Pattern Recognition Letters*, Vol.1, No.4, pp.245-253, 1983.

[7] W. J. Christmas, J. Kittler, and M. Petrou, "Structural matching in computervision using probabilistic relaxation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):749–764, August 1995.

[8] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-Based Technologies for Intelligence Analysis," *Communications of the ACM, Special Issue on Emerging Technologies for Homeland Security*, Vol. 47, No. 3, pp. 45-47, 2004.

[9] D.J. Cook and L.B. Holder, "Substructure Discovery Using Minimum Description Length and Background Knowledge," *J. Artificial Intelligence Research*, Vol. 1, pp. 231-255, Feb. 1994.

[10] T. Darr, S. Greenblatt, and D. Strack, "A Multi-INT Level 2-3 Fusion Framework for Counter-Terrorism," Presented at *Working Together: R&D Partnerships in Homeland Security*, April 27 & 28, 2005.

[11] S. Djoko, D. J. Cook, and L. B. Holder, "An Empirical Study of Domain Knowledge and its Benefits to Substructure Discovery," *IEEE Transactions on Knowledge and Data Engineering*, 9(4), 1997.

[12] P. Foggia, C. Sansone, and M. Vento, "A Performance Comparison of Five Algorithms for Graph Isomorphism," *International Workshop on Graph-based Representation in Pattern Recognition*, Ischia, Italy, pp. 188 - 199, 23 – 25, May , 2001.

[13] L. Getoor, "Learning Statistical Models from Relational Data," Ph.D. Thesis, Stanford University, December, 2001.

[14] P. B. Gibbons and M. Garofalakis, "Approximate query processing: Taming the terabytes!" (tutorial), *Proceedings of VLDB*, 2001.

[15] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," *Proceedings of the International Conference in Pattern recognition (ICPR)*, Quebec, Canada, 2002.

[16] B. Goethals, E. Hoekx, and J. Van den Bussche, "Mining tree queries in a graph," *The Eleventh ACM SIGKDD InternationalCconference on Knowledge Discovery and Data Mining*, 2005.

[17] S. Greenblatt, S. Marcus, and T. Darr, "TMODS - Integrated Fusion Dashboard - Applying Fusion of Fusion Systems to Counter-Terrorism," Presented to the *2005 International Conference on Intelligence Analysis*, May 2-6, 2005.

[18] R. H. Guting, "GraphDB: Modeling and Querying Graphs in Databases," *Proceedings of VLDB'94*, Santiago de Chile, pp. 297—308, Morgan Kaufmann, 1994.

[19] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on Systems Science and Cybernetics* 4(2), 100—107, 1968.

[20] J. Huan, W. Wang., and J. Prins, "Efficient mining of frequent subgraphs in the presence of isomorpism," *Proceedings IEEE International Conference on Data Mining (ICDM)*, 2003.

[21] Y. E. Ioannidis and V. Poosala, "Balancing Histogram Optimality and Practicality for Query Result Size Estimation," *Proceedings of the 1995 ACM SIGMOD International Conference on the Management of Data*, 1995.

[22] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," *SIAM International Conference on Data Mining (SDM-04)*, 2004.

[23] D. Lin, "An information-theoretic definition of similarity," *Proceedings of the 15th International Conference on Machine Learning*, Madison, WI, 1998.

[24] B. D. McKay, "Nauty User's Guide (Version 1.5)," Technical Report TR-CS-9002, Department of Computer Science, Australian National University, Canberra, Australia, 1990.

[25] B. T. Messmer and H. Bunke, "Subgraph Isomorphism in Polynomial Time," Technical Report TR-IAM-95-003, 1995.

[26] S. Nijssen and J. N. Kok, "Frequent Graph Mining and its Application to Molecular Databases" (tutorial), *Proceedings of IEEE Conference on Systems, Mand and Cybernetics (SMC)*, 2004.

[27] N. Polyzotis and M. Garofalakis, "Statistical synopses for graph-structured XML databases," *SIGMOD Conference*, 2002.

[28] R. Rada, H. Mili, E. Bicknell, M. Blettner, "Development and application of a metric on semantic nets," *IEEE Transactions on Systems, Man and Cybernetics*, 1989.

[29] D. Shasha, J. T. L. Wang, and R. Giugno, "Algorithmics and applications of tree and graph searching," *Symposium on Principles of Database Systems*, pages 39--52, 2002.

[30] SRI International, "Link Analysis Workbench," AFRL-IF-RS-TR-2004-247, Final Technical Report, September 2004.

[31] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM* 23(1), 31-42, 1976.

[32] S. Umeyama, "An eigendecomposition approach to weighted graph matching problems," *IEEE Transactions on Pattern Matching and Machine Intelligence*, 10(5):695-703, 1988.

[33] W. Wang, H. Jiang, H. Lu, and J. Xu Yu, "Bloom Histogram: Path Selectivity Estimation for XML Data with Updates," *VLDB*, 2004.

[34] T. Washio and H. Motoda, "State of the Art of Graph-based Data Mining," *SIGKDD Explorations Special Issue on Multi-Relational Data Mining*, Volume 5, Issue 1, 2003.

[35] M. Wolverton, Personal communication with author, March 23, 2006.

[36] M. Wolverton, P. Berry, I. Harrison, J. Lowrance, D. Morley, A. Rodriguez, E. Ruspini, and J. Thomere, "LAW: A Workbench for Approximate Pattern Matching in Relational Data," *Proceedings of the Fifteenth Innovative Applications of Artificial Intelligence Conference (IAAI-03)*, 2003.

[37] M. Wolverton, I. Harrison, J. Lowrance, A. Rodriguez, and J. Thomere, "Software Supported Pattern Development in Intelligence Analysis," 2006. (http://www.ai.sri.com/pubs/files/1147.pdf)

[38] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen, "A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston," *Knowledge Discovery in Database: PKDD 2005 (9th European Conference on Principles and Practices of Knowledge Discovery in Databases, Porto, Portugal 2005-10-03 - 2005-10-07)*, pp. 392-403, Springer, 2005.

[39] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," *Proceedings of IEEE International Conference on Data Mining (ICDM)*, 2002.