Chapter 2

GRAPH DATA MANAGEMENT AND MINING: A SURVEY OF ALGORITHMS AND APPLICATIONS

Charu C. Aggarwal
IBM T. J. Watson Research Center
Hawthorne, NY 10532, USA
charu@us.ibm.com

Haixun Wang
Microsoft Research Asia
Beijing, China 100190
haixunw@microsoft.com

Abstract Graph mining and management has become a popular area of research in recent years because of its numerous applications in a wide variety of practical fields, including computational biology, software bug localization and computer networking. Different applications result in graphs of different sizes and complexities. Correspondingly, the applications have different requirements for the underlying mining algorithms. In this chapter, we will provide a survey of different kinds of graph mining and management algorithms. We will also discuss a number of applications, which are dependent upon graph representations. We will discuss how the different graph mining algorithms can be adapted for different applications. Finally, we will discuss important avenues of future research in the area.

Keywords: Graph Mining, Graph Management

1. Introduction

Graph mining has been a popular area of research in recent years because of numerous applications in computational biology, software bug localization and computer networking. In addition, many new kinds of data such as semi-

C.C. Aggarwal and H. Wang (eds.), *Managing and Mining Graph Data*,
Advances in Database Systems 40, DOI 10.1007/978-1-4419-6045-0_2,

structured data and XML [8] can typically be represented as graphs. A detailed discussion of various kinds of graph mining algorithms may be found in [58].

In the graph domain, the requirement of different applications is not very uniform. Thus, graph mining algorithms which work well in one domain may not work well in another. For example, let us consider the following domains of data:

- **Chemical Data:** Chemical data is often represented as graphs in which the nodes correspond to atoms, and the links correspond to bonds between the atoms. In some cases, substructures of the data may also be used as individual nodes. In this case, the individual graphs are quite small, though there are significant repetitions among the different nodes. This leads to isomorphism challenges in applications such as graph matching. The isomorphism challenge is that the nodes in a given pair of graphs may match in a variety of ways. The number of possible matches may be exponential in terms of the number of the nodes. In general, the problem of isomorphism is an issue in many applications such as frequent pattern mining, graph matching, and classification.

- **Biological Data:** Biological data is modeled in a similar way as chemical data. However, the individual graphs are typically much larger. Furthermore, the nodes are typically carefully designed portions of the biological models. A typical example of a node in a DNA application could be an amino-acid. A single biological network could easily contain thousands of nodes. The sizes of the overall database are also large enough for the underlying graphs to be disk-resident. The disk-resident nature of the data set often leads to unique issues which are not encountered in other scenarios. For example, the access order of the edges in the graph becomes much more critical in this case. Any algorithm which is designed to access the edges in random order will not work very effectively in this case.

- **Computer Networked and Web Data:** In the case of computer networks and the web, the number of nodes in the underlying graph may be massive. Since the number of nodes is massive, this can lead to a very large number of *distinct edges*. This is also referred to as the *massive domain issue* in networked data. In such cases, the number of distinct edges may be so large, that they may be hard to hold in the available storage space. Thus, techniques need to be designed to summarize and work with condensed representations of the graph data sets. In some of these applications, the edges in the underlying graph may arrive in the form of a data stream. In such cases, a second challenge arises from the fact that it may not be possible to store the incoming edges for future analysis. Therefore, the summarization techniques are especially essential for this

case. The stream summaries may be leveraged for future processing of the underlying graphs.

- **XML data:** XML data is a natural form of graph data which is fairly general. We note that mining and management algorithms for XML data are also quite useful for graphs, since XML data can be viewed as labeled graphs. In addition, the attribute-value combinations associated with the nodes makes the problem much more challenging. However, the research in the field of XML data has often been quite independent of the research in the graph mining field. Therefore, we will make an attempt in this chapter to discuss the XML mining algorithms along with the graph mining and management algorithms. It is hoped that this will provide a more integrated view of the field.

It is clear that the design of a particular mining algorithm depends upon the application domain at hand. For example, a disk-resident data set requires careful algorithmic design in which the edges in the graph are not accessed randomly. Similarly, massive-domain networks require careful summarization of the underlying graphs in order to facilitate processing. On the other hand, a chemical molecule which contains a lot of repetitions of node-labels poses unique challenges to a variety of applications in the form of *graph isomorphism*.

In this chapter, we will discuss different kinds of graph management and mining applications, along with the corresponding applications. We note that the boundary between graph mining and management algorithms is often not very clear, since many kinds of algorithms can often be classified as both. The topics in this chapter can primarily be divided into three categories. These categories discuss the following:

- **Graph Management Algorithms:** This refers to the algorithms for managing and indexing large volumes of the graph data. We will present algorithms for indexing of graphs, as well as processing of graph queries. We will study other kinds of queries such as reachability queries as well. We will study algorithms for matching graphs and their applications.

- **Graph Mining Algorithms:** This refers to algorithms used to extract patterns, trends, classes, and clusters from graphs. In some cases, the algorithms may need to be applied to large collections of graphs on the disk. We will discuss methods for clustering, classification, and frequent pattern mining. We will also provide a detailed discussion of these algorithms in the literature.

- **Applications of Graph Data Management and Mining:** We will study various application domains in which graph data management and mining algorithms are required. This includes web data, social and computer networking, biological and chemical data, and software bug localization.

This chapter is organized as follows. In the next section, we will discuss a variety of graph data management algorithms. In section 3, we will discuss algorithms for mining graph data. A variety of application domains in which these algorithms are used is discussed in section 4. Section 5 discusses the conclusions and summary. Future research directions are discussed in the same section.

2. Graph Data Management Algorithms

Data management of graphs has turned out to be much more challenging than that for multi-dimensional data. The structural representation of graphs has greater expressive power, but it comes at a cost. This cost is in terms of the complexity of data representation, access, and processing, because intermediate operations such as similarity computations, averaging, and distance computations cannot be naturally defined for structural data in as intuitive a way as is the case for multidimensional data. Furthermore, traditional relational databases can be efficiently accessed with the use of block read-writes; this is not as natural for structural data in which the edges may be accessed in arbitrary order. However, recent advances have been able to alleviate some of these concerns at least partially. In this section, we will provide a review of many of the recent graph management algorithms and applications.

2.1 Indexing and Query Processing Techniques

Existing database models and query languages, including the relational model and SQL, lack native support for advanced data structures such as trees and graphs. Recently, due to the wide adoption of XML as the de facto data exchange format, a number of new data models and query languages for tree-like structures have been proposed. More recently, a new wave of applications across various domains including web, ontology management, bioinformatics, etc., call for new data models, languages and systems for graph structured data.

Generally speaking, the task can be simple put as the following: For a query pattern (a tree or a graph), find graphs or trees in the database that contain or are similar to the query pattern. To accomplish this task elegantly and efficiently, we need to address several important issues: i) how to model the data and the query; ii) how to store the data; and iii) how to index the data for efficient query processing.

Query Processing of Tree Structured Data. Much research has been done on XML query processing. On a high level, there are two approaches for modeling XML data. One approach is to leverage the existing relational model after mapping tree structured data into relational schema [169]. The other approach is to build a native XML database from scratch [106]. For

instance, some works starts with creating a tree algebra and calculus for XML data [107]. The proposed tree algebra extends the relational algebra by defining new operators, such as node deletion and insertion, for tree structured data.

SQL is the standard access method for relational data. Much efforts have been made to design SQL's counterpart for tree structured data. The criteria are, first expressive power, which allows users the flexibility to express queries over tree structured data, and second declarativeness, which allows the system to optimize query processing. The wide adoption of XML has spurred standards body groups to expand the SQL specification to include XML processing functions. XQuery [26] extends XPath [52] by using a `FLWOR`[1] structure to express a query. The `FLWOR` structure is similar to SQL's `SELECT-FROM-WHERE` structure, with additional support for iteration and intermediary variable binding. With path expressions and the `FLWOR` construct, XQuery brings SQL-like query power to tree structured data, and has been recommended by the World Wide Web Consortium (W3C) as the query language for XML documents.

For XML data, the core of query processing lies in efficient tree pattern matching. Many XML indexing techniques have been proposed [85, 141, 132, 59, 51, 115] to support this operation. DataGuide [85], for example, provides a concise summary of the path structure in a tree-structured database. T-index [141], on the other hand, indexes a specific set of path expressions. Index Fabric [59] is conceptually similar to DataGuide in that it keeps all label paths starting from the root element. Index Fabric encodes each label path to each XML element with a data value as a string and inserts the encoded label path and data value into an index for strings such as the Patricia tree. APEX [51] uses data mining algorithms to find paths that appear frequently in query workload. While most techniques focused on simple path expressions, the F^+B Index [115] emphasizes on branching path expressions (twigs). Nevertheless, since a tree query is decomposed into node, path, or twig queries, joining intermediary results together has become a time consuming operation. Sequence-based XML indexing [185, 159, 186] makes tree patterns a first class citizen in XML query processing. It converts XML documents as well as queries to sequences and performs tree query processing by (non-contiguous) subsequence matching.

Query Processing of Graph Structured Data. One of the common characteristics of a wide range of nascent applications including social networking, ontology management, biological network/pathways, etc., is that the data they are concerned with is all graph structured. As the data increases in size and complexity, it becomes important that it is managed by a database system.

There are several approaches to managing graphs in a database. One possibility is to extend a commercial RDBMS engine to support graph structured data. Another possibility is to use general purpose relational tables to store

graphs. When these approaches fail to deliver needed performance, recent research has also embraced the challenges of designing a special purpose graph database. Oracle is currently the only commercial DBMS that provides internal support for graph data. Its new 10g database includes the Oracle Spatial network data model [3], which enables users to model and manipulate graph data. The network model contains logical information such as connectivity among nodes and links, directions of links, costs of nodes and links, etc. The logical model is mainly realized by two tables: a node table and a link table, which store the connectivity information of a graph. Still, many are concerned that the relational model is fundamentally inadequate for supporting graph structured data, for even the most basic operations, such as graph traversal, are costly to implement on relational DBMSs, especially when the graphs are large. Recent interest in Semantic Web has spurred increased attention to the Resource Description Framework (RDF) [139]. A *triplestore* is a special purpose database for the storage and retrieval of RDF data. Unlike a relational database, a triplestore is optimized for the storage and retrieval of a large number of short statements in the form of subject-predicate-object, which are called triples. Much work has been done to support efficient data access on the triplestore [14, 15, 19, 33, 91, 152, 182, 195, 38, 92, 194, 193]. Recently, the semantic web community has announced the billion triple challenge [4], which further highlights the need and urgency to support inferencing over massive RDF data.

A number of graph query languages have been proposed since early 1990s. For example, GraphLog [56], which has its roots in Datalog, performs inferencing on rules (possibly with negation) about graph paths represented by regular expressions. GOOD [89], which has its roots in object-oriented databases, defines a transformation language that contains five basic operations on graphs. GraphDB [88], another object-oriented data model and query language for graphs, performs queries in four steps, each carrying out operations on subgraphs specified by regular expressions. Unlike previous graph query languages that operate on nodes, edges, or paths, GraphQL [97] operates directly on graphs. In other words, graphs are used as the operand and return type of all operations. GraphQL extends the relational algebraic operators, including selection, Cartesian product, and set operations, to graph structures. For instance, the selection operator is generalized to graph pattern matching. GraphQL is relationally complete and the nonrecursive version of GraphQL is equivalent to the relational algebra. A detailed description of GraphQL and a comparison of GraphQL with other graph query languages can be found in [96].

With the rise of Semantic Web applications, the need to efficiently query RDF data has been propelled into the spotlight. The SPARQL query language [154] is designed for this purpose. As we mentioned before, a graph in the RDF format is described by a set of triples, each corresponding to an edge between two nodes. A SPARQL query, which is also SQL-like, may con-

sist of triple patterns, conjunctions, disjunctions, and optional patterns. A triple pattern is syntactically close to an RDF triple except that each of the subject, predicate and object may be a variable. The SPARQL query processor will search for sets of triples that match the triple patterns, binding the variables in the query to the corresponding parts of each triple [154].

Another line of work in graph indexing uses important structural characteristics of the underlying graph in order to facilitate indexing and query processing. Such structural characteristics can be in the form of paths or frequent patterns in the underlying graphs. These can be used as *pre-processing filters*, which remove irrelevant graphs from the underlying data at an early stage. For example, the *GraphGrep* technique [83] uses the enumerated paths as index features which can be used in order to filter unmatched graphs. Similarly, the *GIndex* technique [201] uses discriminative frequent fragments as index features. A closely related technique [202] leverages on the substructures in the underlying graphs in order to facilitate indexing. Another way of indexing graphs is to use the tree structures [208] in the underlying graph in order to facilitate search and indexing.

The topic of query processing on graph data has been studied for many years, still, many challenges remain. On the one hand, data is becoming increasingly large. One possibility of handling such large data is through parallel processing, by using for example, the Map/Reduce framework. However, it is well known that many graph algorithms are very difficult to be parallelized. On the other hand, graph queries are becoming increasingly complicated. For example, queries against a complex ontology are often lengthy, no matter what graph query language is used to express the queries. Furthermore, when querying a complex graph (such as a complex ontology), users often have only a vague notion, rather than a clear understanding and definition, of what they query for. These call for alternative methods of expressing and processing graph queries. In other words, instead of explicitly expressing a query in the most exact terms, we might want to use keyword search to simplify queries [183], or using data mining methods to semi-automate query formation [134].

2.2 Reachability Queries

Graph reachability queries test whether there is a path from a node v to another node u in a large directed graph. Querying for reachability is a very basic operation that is important to many applications, including applications in semantic web, biology networks, XML query processing, etc.

Reachability queries can be answered by two obvious methods. In the first method, we traverse the graph starting from node v using breath- or depth-first search to see whether we can ever reach node u. The query time is $O(n + m)$,

where n is the number of nodes and m is the number of edges in the graph. At the other extreme, we compute and store the edge transitive closure of the graph. With the transitive closure, which requires $O(n^2)$ storage, a reachability query can be answered in $O(1)$ time by simply checking whether (u, v) is in the transitive closure. However, for large graphs, neither of the two methods is feasible: the first method is too expensive at query time, and the second takes too much space.

Research in this area focuses on finding the best compromise between the $O(n + m)$ query time and the $O(n^2)$ storage cost. Intuitively, it tries to compress the reachability information in the transitive closure and answer queries using the compressed data.

Spanning tree based approaches. Many approaches, for example [47, 176, 184], decompose a graph into two parts: i) a spanning tree, and ii) edges not on the spanning tree (non-tree edges). If there is a path on the spanning tree between u and v, reachability between u and v can be decidedly easily. This is done by assigning each node u an interval code (u_{start}, u_{end}), such that v is reachable from u if and only if $u_{start} \leq v_{start} \leq u_{end}$. The entire tree can be encoded by performing a simple depth-first traversal of the tree. With the encoding, reachability check can be done in $O(1)$ time.

If the two nodes are not connected by any path on the spanning tree, we need to check if there is a path that involves non-tree edges connecting the two nodes. In order to do this, we need to build index structures in addition to the interval code to speed up the reachability check. Chen et al. [47] and Trißl et al. [176] proposed index structures for this purpose, and both of their approaches achieve $O(m - n)$ query time. For instance, Chen et al.'s SSPI (Surrogate & Surplus Predecessor Index) maintains a predecessor list $PL(u)$ for each node u, which, together with the interval code, enables efficient reachability check. Wang et al. [184] made an observation that many large graphs in real applications are sparse, which means the number of non-tree edges is small. The algorithm proposed based on this assumption answers reachability queries in O(1) time using a $O(n + t^2)$ size index structure, where t is the number of non-tree edges, and $t \ll n$.

Set covering based approaches. Some approaches propose to use simpler data structures (e.g., trees, paths, etc) to "cover" the reachability information embodied by a graph structure. For example, if v can reach u, then v can reach any node in a tree rooted at u. Thus, if we include the tree in the index, we cover a large set of reachability in the graph. We then use multiple trees to cover an entire graph. Agrawal et al. [10]'s optimal tree cover achieves $O(\log n)$ query time, where n is the number of nodes in the graph. Instead of using trees, Jagadish et al. [105] proposes to decompose a graph into pairwise

disjoint *chains*, and then use chains to cover the graph. The intuition of using a chain is similar to using a tree: if v can reach u on a chain, then v can reach any node that comes after u on that chain. The chain-cover approach achieves $O(nk)$ query time, where k is the number of chains in the graph. Cohen et al. [54] proposed a 2-hop cover for reachability queries. A node u is labeled by two sets of nodes, called $L_{in}(u)$ and $L_{out}(u)$, where $L_{in}(u)$ are the nodes that can reach u and $L_{out}(u)$ are the ones that u can reach. The 2-hop approach assigns the L_{in} and L_{out} labels to each node such that u can reach v if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The optimal 2-hop cover problem of finding the minimum size 2-hop cover is NP-hard. A greedy algorithm finds a 2-hop cover iteratively. In each iteration, it picks the node w that maximizes the value of $\frac{S(A_w,w,D_w) \cap TC'}{|A_w|+|D_w|}$, where $S(A_w, w, D_w) \cap TC'$ represents the new (uncovered) reachability that a 2-hop cluster centered at w can cover, and $|A_w| + |D_w|$ is the cost (size) of the 2-hop cluster centered at w. Several algorithms have been proposed to compute high quality 2-hop covers [54, 168, 49, 48] in a more efficient manner. Many extensions to existing set covering based approaches have been proposed. For example, Jin et al. [112] introduces a 3-hop cover approach that combines the chain cover and the 2-hop cover.

Extensions to the reachability problem. Reachability queries are one of the most basic building blocks for many advanced graph operations, and some are directly related to reachability queries. One interesting problem is in the domain of labeled graphs. In many applications, edges are labeled to denote the relationships between the two nodes they connect. A new type of reachability query asks whether two nodes are connected by a path whose edges are constrained by a given set of labels [111]. In some other applications, we want to find the shortest path between two nodes. Similar to the simple reachability problem, the shortest path problem can be solved by brute force methods such as Dijkstra's algorithm, but such methods are not appropriate for online queries in large graphs. Cohen et al extended the 2-hop covering approach for this problem [54].

A detailed description of the strengths and weaknesses of various reachability approaches and a comparison of their query time, index size, and index construction time can be found in [204].

2.3 Graph Matching

The problem of graph matching is that of finding either an approximate or a one-to-one correspondence among the nodes of the two graphs. This correspondence is based on one or more of the following structural characteristics of the graph: (1) The labels on the nodes in the two graphs should be the same. (2) The existence of edges between corresponding nodes in the two graphs

should match each other. (3) The labels on the edges in the two graphs should match each other.

These three characteristics may be used to define a matching between two graphs such that there is a one-to-one correspondence in the structures of the two graphs. Such problems often arise in the context of a number of different database applications such as schema matching, query matching, and vector space embedding. A detailed description of these different applications may be found in [161]. In *exact graph matching*, we attempt to determine a one-to-one correspondence between two graphs. Thus, if an edge exists between a pair of nodes in one graph, then that edge must also exist between the corresponding pair in the other graph. This may not be very practical in real applications in which *approximate matches may exist*, but an exact matching may not be feasible. Therefore, in many applications, it is possible to define an objective function which determines the similarity in the mapping between the two graphs. Fault tolerant mapping is a much more significant application in the graph domain, because common representations of graphs may have many missing nodes and edges. This problem is also referred to as *inexact graph matching*. Most variants of the graph matching problem are well known to be NP-hard. The most common method for graph matching is that of tree-based search techniques. In this technique, we start with a seed set of nodes which are matched, and iteratively expand the neighborhood defined by that set. Iterative expansion can be performed by adding nodes to the current node set, as long as no edge constraints are violated. If it turns out that the current node set cannot be expanded, then we initiate a backtracking procedure in which we undo the last set of matches. A number of algorithms which are based upon this broad idea are discussed in [60, 125, 180]. A survey of many of the classical algorithms for graph matching may be found in [57].

The problem of exact graph matching is closely related to that of graph isomorphism. In the case of the graph isomorphism problem, we attempt to find an exact one-to-one matching between nodes and edges of the two graphs. A generalization of this problem is that of finding the maximal common subgraph in which we attempt to match the maximum number of nodes between the two graphs. Note that the solution to the maximal common subgraph problem will also provide a solution to the problem of exact matching between two subgraphs, if such a solution exists. A number of similarity measures can be derived on the basis of the mapping behavior between two graphs. If the two graphs share a large number of nodes in common, then the similarity is more significant. A number of models and algorithms for quantifying and determining the common subgraphs between two graphs may be found in [34–37]. The broad idea in many of these methods is to define a distance metric based on the nature of the matching between the two graphs, and use this distance metric in order to guide the algorithms towards an effective solution.

Inexact graph matching is a much more practical model, because it accounts for the natural errors which may occur during the matching process. Clearly, a method is required in order to quantify these errors and the closeness between the different graphs. A common technique which may be used to quantify these errors is the use of a function such as the graph edit distance. The graph edit distance determines the distance between two graphs by measuring the *cost of the edits required* to transform one graph to the other. These edits may be node or edge insertions, deletions or substitutions. An *inexact graph matching* is one which allows for a matching between two graphs after a sequence of such edits. The quality of the matching is defined by the cost of the corresponding edits. We note that the concept of graph edit distance is closely related to that of finding a maximum common subgraph [34]. This is because it is possible to direct an edit-distance based algorithm to find the maximum common subgraph by defining an appropriate edit distance.

A particular variant of the problem is when we account for the values of the labels on the nodes and edges during the matching process. In this case, we need to compute the distance between the labels of the nodes and edges in order to define the cost of a label substitution. Clearly, the cost of the label substitution is *application-dependent*. In the case of numerical labels, it may be natural to define the distances based on numerical distance functions between the two graphs. In general, the cost of the edits is also application dependent, since different applications may use different notions of similarity. Thus, domain-specific techniques are often used in order to define the edit costs. In some cases, the edit costs may even be learned with the use of sample graphs [143, 144]. When we have cases in which the sample graphs have naturally defined distances between them, the edit costs may be determined as values for which the corresponding distances are as close to the sample values as possible.

The typical algorithms for inexact graph matching use combinatorial search over the space of possible edits in order to determine the optimal matching [35, 145]. The algorithm in [35] is relatively exhaustive in its approach, and can therefore be computationally intensive in practice. In order to solve this issue, the algorithms discussed in [145] explores local regions of the graph in order to define more focussed edits. In particular, the work in [145] proposes an important class of methods which are referred to as *kernel functions*. Such methods are extremely robust to structural errors, and are therefore a useful construct for solving graph matching problems. The broad idea is to incorporate the key ideas of the graph edit distance into kernel functions. Since kernel machines are known to be extremely powerful techniques for pattern recognition, it follows that these techniques can then be leveraged to the problem of graph matching. A variety of other kernel techniques for graph matching may be found in [94, 81, 119]. The key kernel methods include convolution kernels

[94], random walk kernels [81] and diffusion kernels [119]. In random walk kernels [81], we attempt to determine the number of random walks between the two graphs which have some labels in common. Diffusion kernels [119] can be considered a generalization of the standard gaussian kernel in Euclidian space.

The technique of *relaxation labeling* is another broad class of methods which is often used for graph matching. Note that in the case of the matching problem, we are really trying to assign labels to the nodes in a graph. The specific label for a node is drawn out of a discrete set of possibilities. This discrete set of possibilities correspond to the matching nodes in the other graph. The probability of matching is defined by Gaussian probability distributions. We start off with an initial labeling based on the structural characteristics of the underlying graph, and then successively improve the solution based on additional exploration of structural information. Detailed descriptions of techniques for relaxation labeling may be found in [76].

2.4 Keyword Search

In the problem of keyword search, we would like to determine small groups of link-connected nodes which are related to a particular keyword. For example, a web graph or a social network may be considered a massive graph, in which each node may contain a large amount of text data. Even though keyword search is defined with respect to the text inside the nodes, we note that the linkage structure also plays an important role in determining the appropriate set of nodes. It is well known the text in linked entities such as the web are related, when the corresponding objects are linked. Thus, by finding groups of closely connected nodes which share keywords, it is generally possible to determine the qualitatively effective nodes. Keyword search provides a simple but user-friendly interface for information retrieval on the Web. It also proves to be an effective method for accessing structured data. Since many real life data sets are structured as tables, trees and graphs, keyword search over such data has become increasingly important and has attracted much research interest in both the database and the IR communities.

Graph is a general structure and it can be used to model a variety of complex data, including relational data and XML data. Because the underlying data assumes a graph structure, keyword search becomes much more complex than traditional keyword search over documents. The challenges lie in three aspects:

- **Query semantics**. Keyword search over a set of text documents has very clear semantics: A document satisfies a keyword query if it contains every keyword in the query. In our case, the entire dataset is often considered as a single graph, so the algorithms must work on a finer granularity

and return subgraphs as answers. We must decide what subgraphs are qualified as answers.

- **Ranking strategy**: For a given keyword query, it is likely that many subgraphs will satisfy the query, based on the query semantics in use. However, each subgraph has its own underlying graph structure, with subtle semantics that makes it different from other subgraphs that satisfy the query. Thus, we must take the graph structure into consideration and design ranking strategies that find most meaningful and relevant answers.

- **Query efficiency**: Many real life graphs are extremely large. A major challenge for keyword search over graph data is query efficiency, which, to a large extent, hinges on the semantics of the query and the ranking strategy.

Current approaches for keyword search can be classified into three categories based on the underlying structure of the data. In each category, we briefly discuss query semantics, ranking strategies, and representative algorithms.

Keyword search over XML data. XML data is mostly tree structured, where each node only has a single incoming path. This property has significant impact on query semantics and answer ranking, and it also provides great optimization opportunities in algorithm design [197].

Given a query, which contains a set of keywords, the search algorithm returns snippets of an XML document that are most relevant to the keywords. The interpretation of *relevant* varies, but the most common practice is to find smallest subtrees that contain the keywords.

It is straightforward to find subtrees that contain all the keywords. Let L_i be the set of nodes in the XML document that contain keyword k_i. If we pick one node n_i from each L_i, and form a subtree from these nodes, then the subtree will contain all the keywords. Thus, an answer to the query can be represented by $lca(n_1, \cdots, n_n)$, the lowest common ancestor of nodes $n_1, \cdots, n_n$ in the tree, where $n_i \in L_i$.

Most query semantics are only interested in *smallest* answers. There are different ways to interpret the notion of *smallest*. Several algorithms [197, 102, 196] are based on the SLCA (smallest lowest common ancestor) semantics, which requires that an answer (a least common ancestor of nodes that contain all the keywords) does not have any descendent that is also an answer. XRank [86] adopts a different query semantics for keyword search. In XRank, answers consist of substrees that contain at least one occurrence of all of the query keywords, after excluding the sub-nodes that already contain all of the

query keywords. Thus, the set of answers based on the SLCA semantics is a subset of answers qualified for XRank.

A keyword query may find a large number of answers, but they are not all equal due to the differences in the way they are embedded in the nested XML structure. Many approaches for keyword search on XML data, including XRank [86] and XSEarch [55], present a ranking method. A ranking mechanism takes into consideration several factors. For instance, more specific answers should be ranked higher than less specific answers. Both SLCA and the semantics adopted by XRank signify this consideration. Furthermore, keywords in an answer should appear *close* to each other, and closeness is interpreted as the the semantic distance defined over the XML embedded structure.

Keyword search over relational data. SQL is the de-facto query language for accessing relational data. However, to use SQL, one must have knowledge about the schema of the relational data. This has become a hindrance for potential users to access tremendous amount of relational data.

Keyword search is a good alternative due to its ease of use. The challenges of applying keyword search on relational data come from the fact that in a relational database, information about a single entity is usually divided among several tables. This is resulted from the normalization principle, which is the design methodology of relational database schema.

Thus, to find entities that are relevant to a keyword query, the search algorithm has to join data from multiple tables. If we represent each table as a node, and each foreign key relationship as an edge between two nodes, then we obtain a graph, which allows us to convert the current problem to the problem of keyword search over graphs. However, there is the possibility of self-joins: that is, a table may contain a foreign key that references itself. More generally, there might be cycles in the graph, which means the size of the join is only limited by the size of the data. To avoid this problem, the search algorithm may adopt an upper bound to restrict the number of joins [103].

Two most well-known keyword search algorithm for relational data are DBXplorer [12] and DISCOVER [103]. They adopted new physical database design (including sophisticated indexing methods) to speed up keyword search over relational databases. Qin et al [155], instead, introduced a method that takes full advantage of the power of RDBMS and uses SQL to perform keyword search on relational data.

Keyword search over graph data. Keyword search over large, schema-free graphs faces the challenge of how to efficiently explore the graph structure and find subgraphs that contain all the keywords in the query. To measure the "goodness" of an answer, most approaches score each edge and node, and then aggregate the scores over the subgraph as a goodness measure [24, 113, 99].

Usually, an edge is scored by the strength of the connection, and a node is scored by its importance based on a PageRank like mechanism.

Graph keyword search algorithms can be classified into two categories. Algorithms in the first category finds matching subgraphs by exploring the graph link by link, without using any index of the graph. Representative algorithms in this category include BANKS [24] and the bidirectional search algorithm [113]. One drawback of these approaches is that they explore the graph blindly as they do not have a global picture of the graph structure, nor do they know the keyword distribution in the graph. Algorithms in the other category are index-based [99], and the index is used to control guide the graph exploration, and support forward-jumps in the search.

2.5 Synopsis Construction of Massive Graphs

A key challenge which arises in many of the applications discussed below is that the graphs they deal with are very large scale in nature. As a result, the graph may be available only on disk. Most of the traditional graph mining applications assume that the data is available in main memory. However, when the graph is available on disk, applications which access the edges in random order may be extremely expensive. For example, the problem of finding the minimum-cut between two nodes is extremely efficient with the use of memory resident algorithms, but it is extraordinarily expensive when the underlying graphs are available on disk [7]. As a result algorithms need to be carefully designed in order to reduce the disk-access costs. A typical technique which may often be used is to design a synopsis construction technique [7, 46, 142], which summarizes the graph in a much smaller space, but retains sufficient information in order to effectively respond to queries.

The synopsis construction is typically defined through either node or edge contractions. The key is to define a synopsis which retains the relevant structural property of the underlying graph. In [7], the algorithm in [177] is used in order to collapse the dense regions of the graph, and represent the summarized graph in terms of sparse regions. The resulting contracted graph still retains important structural properties such as the connectivity of the graph. In [46], a randomized summarization technique is used in order to determine frequent patterns in the underlying graph. A bound has been proposed in [46] for determining the false positives and false negatives with the use of this approach. Finally, the technique in [142] also compresses graphs by representing sets of nodes as super-nodes, and separately storing "edge corrections" in order to reconstruct the entire graph. A bound on the error has been proposed in [142] with the use of this approach.

A closely related problem is that of mining *graph streams*. In this case, the edges of the graph are received continuously over time. Such cases arise

frequently in applications such as social networks, communication networks, and web log analysis. Graph streams are very challenging to mine, because the structure of the graph needs to be mined in real time. Therefore, a typical approach is to construct a synopsis from the graph stream, and leverage it for the purpose of structural analysis. It has been shown in [73] how to summarize the graph in such a way that the underlying distances are preserved. Therefore, this summarization can be used for distance-based applications such as the shortest path problem. A second application which has been studied in the context of graph streams is that of *graph matching* [140]. We note that this is a different version of the problem from our discussion in an earlier section. In this case, we attempt to find a set of edges in a single graph such that no two edges share an end point. We desire to find a maximum weight or maximum cardinality matching. The main idea in [140] is to always maintain a candidate matching and update it as new edges come in. When a new edge arrives, the process of inserting it may displace as many as two edges at its end points. We allow an incoming edge to displace the edges at its endpoints, if the weight of the incoming edge is a factor $(1 + \gamma)$ of the outgoing edges. It has been shown in [140] that this matching is within a factor $(3 + 2 \cdot \sqrt{2})$ of the optimal matching.

Recently, a number of techniques have also been designed to create synopses which can be used to estimate the aggregate structural properties of the underlying graphs. A technique has been proposed in [61] for estimating the statistics of the degrees in the underlying graph stream. The techniques proposed in [61] use a variety of techniques such as sketches, sampling, hashing and distinct counting. Methods have been proposed for determining the moments of the degrees, determining heavy hitter degrees, and determining range sums of degrees. In addition, techniques have been proposed in [18] to perform space-efficient reductions in data streams. This reduction has been used in order to count triangles in the data stream. A particularly useful application in graph streams is that of the problem of *PageRank*. In this problem, we attempt to determine significant pages in a collection with the use of the linkage structure of the underlying documents. Clearly, documents which are linked to by a larger number of documents are more significant [151]. In fact, the concept of page rank can be modeled as the probability that a node is visited by a random surfer on the world wide web. The algorithms designed in [151] are for static graphs. The problem becomes much more challenging when the graphs are dynamic, as is the case of social networks. A natural synopsis technique which can be used for such cases is the method of sampling. In [166], it has been shown how to use a sampling technique in order to estimate page rank for graph streams. The idea is to sample the nodes in the graph independently and perform random walks starting from these nodes. These random walks can be

used in order to estimate the probability of the presence of a random surfer at a given node. This is essentially equal to the page rank.

3. Graph Mining Algorithms

Many of the traditional mining applications also apply to the case of graphs. As in the case of management applications, the mining applications are far more challenging to implement because of the additional constraints which arise from the structural nature of the underlying graph. In spite of these challenges, a number of techniques have been developed for traditional mining problems such as frequent pattern mining, clustering, and classification. In this section, we will provide a survey of many of the structural algorithms for graph mining.

3.1 Pattern Mining in Graphs

The problem of frequent pattern mining has been widely studied in the context of mining transactional data [11, 90]. Recently, the techniques for frequent pattern mining have also been extended to the case of graph data. The main difference in the case of graphs is that the process of determining support is quite different. The problem can be defined in different ways depending upon the application domain:

- In the first case, we have a group of graphs, and we wish to determine all patterns which support a fraction of the corresponding graphs [104, 123, 181].
- In the second case, we have a single large graph, and we wish to determine all patterns which are supported at least a certain number of times in this large graph [31, 75, 123].

In both cases, we need to account for the isomorphism issue in determining whether one graph is supported by another. However, the problem of defining the support is much more challenging, if overlaps are allowed between different embeddings. This is because if we allow such overlaps, then the anti-monotonicity property of most frequent pattern mining algorithms is violated.

For the first case, where we have a data set containing multiple graphs, most of the well known techniques for frequent pattern mining with transactional data can be easily extended. For example, *Apriori*-style algorithms can be extended to the case of graph data, by using a similar level-wise strategy of generating $(k+1)$-candidates from k-patterns. The main difference is that we need to define the join process a little differently. Two graphs of size k can be joined, if they have a structure of size $(k-1)$ in common. The *size of this structure* could be defined in terms of either nodes or edges. In the case of the AGM algorithm [104], this common structure is defined in terms of

the number of common vertices. Thus, two graphs with k vertices are joined, only if they have a common subgraph with at least $(k-1)$ vertices. A second way of performing the mining is to join two graphs which have a subgraph containing at least $(k-1)$ edges in common. The FSG algorithm proposed in [123] can be used in order to perform edge-based joins. It is also possible to define the joins in terms of arbitrary structures. For example, it is possible to express the graphs in terms of edge-disjoint paths. In such cases, subgraphs with $(k+1)$-edge disjoint paths can be generated from two graphs which have k edge disjoint paths, of which $(k-1)$ must be common. An algorithm along these lines is proposed in [181]. Another strategy which is often used is that of *pattern growth techniques*, in which frequent graph patterns are extended with the use of additional edges [28, 200, 100]. As in the case of frequent pattern mining problem, we use lexicographic ordering among edges in order to structure the search process, so that a given pattern is encountered only once.

For the second case in which we have a single large graph, a number of different techniques may be used in order to define the support in presence of the overlaps. A common strategy is to use the size of the maximum independent set of the overlap graph to define the support. This is also referred to as the *maximum independent set support*. In [124], two algorithms HSIGRAM and VSIGRAM are proposed for determining the frequent subgraphs within a single large graph. In the former case, a breadth-first search approach is used in order to determine the frequent subgraphs, whereas a depth-first approach is used in the latter case. In [75], it has been shown that the maximum independent set measure continues to satisfy the anti-monotonicity property. The main problem with this measure is that it is extremely expensive to compute. Therefore, the technique in [31] defines a different measure in order to compute the support of a pattern. The idea is to compute a *minimum image based support* of a given pattern. For this case, we compute the number of unique nodes of the graph to which a node of the given pattern is mapped. This measure continues to satisfy the anti-monotonicity property, and can therefore be used in order to determine the underlying frequent patterns. An efficient algorithm with the use of this measure has been proposed in [31].

As in the case of standard frequent pattern mining, a number of variations are possible for the case of finding graph patterns, such as determining maximal patterns [100], closed patterns [198], or significant patterns [98, 157, 198]. We note that significant graph patterns can be defined in different ways depending upon the application. In [157], significant graphs are defined by transforming regions of the graphs into features and measuring the corresponding importance in terms of p-values. In [198], significant patterns are defined in terms of arbitrary objective functions. A meta-framework has been proposed in [198] to determine the significant patterns based on arbitrary objective functions. One interesting approach to discover significant patterns is to build a

model-based search tree or MbT[71]. The idea is to use divide and conquer to mine the most significant patterns in a subspace of examples. It builds a decision tree that partitions the data onto different nodes. Then at each node, it directly discovers a discriminative pattern to further divide its examples into purer subsets. Since the number of examples towards leaf level is relatively small, this approach is able to examine patterns with extremely low global support that could not be enumerated on the whole data set. For some graph data sets which occur in drug discovery applications[71], it could mine significant graph patterns, which is very difficult for most other solutions. Since it uses the divide and conquer paradigm, the algorithm is almost linearly scalable with $1 - MinSupport$ and the number of examples[71]. The MbT technique is not limited to graphs, but also applicable to item sets and sequences, and mine pattern set is both small and significant.

One of the key challenges which arises in the context of all frequent pattern mining algorithms is the massive number of patterns which can be mined from the underlying database. This problem is particularly acute in the case of graphs since the size of the output can be extremely large. One solution for reducing the number of representative patterns is to report frequent patterns in terms of *orthogonality*. A model called *ORIGAMI* has been proposed in [93] which reports frequent graph patterns only if the similarity is below a threshold α. Such patterns are also referred to as α*-orthogonal patterns*. A pattern set P is said to be β*-representative*, if for every non-reported pattern g, at least one pattern can be found in P for which the underlying similarity to g is at least a threshold β. These two constraints address different aspects of the structural patterns. The method in [93] determines the set of all α-orthogonal and β-representative patterns. An efficient algorithm has been proposed in [93] in order to mine such patterns. The idea here is to reduce the redundancy in the underlying pattern set so as to provide a better understanding of the reported patterns.

Some particularly challenging variations of the problem arise in the context of either very large data sets or very large data graphs. Recently, a technique was proposed by [46], which uses randomized summarization in order to reduce the data set to a much smaller size. This summarization is then leveraged in order to determine the frequent subgraph patterns from the data. Bounds are derived in [46] on the false positives and false negatives with the use of such an approach. Another challenging variation is when the frequent patterns are overlaid on a very large graph, as a result of which patterns may themselves be very large subgraphs. An algorithm called *TSMiner* was proposed in [110] to determine frequent structures in very large scale graphs.

Graph pattern mining has numerous applications for a variety of applications. For example, in the case of labeled data, such pattern mining techniques can be used in order to determine *structural classification rules*. For example,

the technique in [205] uses this approach for the purpose of XML data classification. In this case, we have a data set consisting of multiple (XML) graphs, each of which is associated with a class label. The method in [205] determines the rules in which the left hand side is a structure and the right hand side is a class label. This is used for the purposes of classification. Another application of frequent pattern mining is studied in [121], in which these patterns are used in order to create $gBoost$, which is a classifier designed as an application of boosting. Frequent pattern mining has been found to be particularly useful in the chemical and biological domain [28, 65, 101, 120]. Frequent pattern mining techniques have been used to perform important functions in this domain such as classification or determination of metabolic pathways.

Frequent graph pattern mining is also useful for the purpose of creating graph indexes. In [201], the frequent structures in a graph collection are mined, so that they can be used as features for an indexing process. The similarity of frequent pattern membership behavior across graphs is used to define a rough similarity function for the purpose of filtering. An inverted representation is constructed on this feature based representation in order to filter out irrelevant graphs for the similarity search process. The technique of [201] is much more efficient than other competitive techniques because of its feature based approach. In general, frequent pattern mining algorithms are useful for any application which can be defined effectively on the basis of aggregate characteristics. In general graph pattern mining techniques have the same range of applicability as they do for the case of vanilla frequent pattern mining.

3.2 Clustering Algorithms for Graph Data

In this section, we will discuss a variety of algorithms for clustering graph data. This includes both classical graph clustering algorithms as well as algorithms for clustering XML data. Clustering algorithms have significant applications in a variety of graph scenarios such as congestion detection, facility location, and XML data integration [126]. Within the context of graph algorithms, the clustering can be of two types:

- **Node Clustering Algorithms:** In this case, we have one large graph, and we attempt to cluster the underlying nodes with the use of a distance (or similarity) value on the edges. In this case, the edges of the graph are labeled with numerical distance values. These numerical distance values are used in order to create clusters of nodes. A particular case is one in which the presence of an edge refers to a similarity value of 1, whereas the absence of an edge refers to a similarity value of 0. We note that the problem of minimizing the inter-cluster similarity for a fixed number of clusters essentially reduces to the problem of *graph partitioning* or the *minimum multi-way cut problem*. This is also referred to as the prob-

lem of mining dense graphs and pseudo-cliques. Recently, the problem has also been studied in the database literature as that of *quasi-clique determination*. In this problem, we determine groups of nodes which are "almost cliques". In other words, an edge exists between any pair of nodes in the set with high probability. We will study the different classes of node clustering algorithms in a different section.

- **Graph Clustering Algorithms:** In this case, we have a (possibly large) number of graphs which need to be clustered based on their underlying structural behavior. This problem is challenging because of the need to match the structures of the underlying graphs, and use these structures for clustering purposes. Such algorithms are discussed both in the context of classical graph data sets as well as semi-structured data. Therefore, we will discuss both of these variations.

In the following subsections, we will discuss each of the above kinds of graph clustering algorithms.

Node Clustering Algorithms. A number of algorithms for graph node clustering are discussed in [78]. In [78], the graph clustering problem is related to the minimum cut and graph partitioning problems. In this case, it is assumed that the underlying graphs have weights on the edges. It is desired to partition the graph in such a way so as to minimize the weights of the edges across the partitions. The simplest case is the 2-way minimum cut problem, in which we wish to partition the graph into two clusters, so as to minimize the weight of the edges across the partitions. This version of the problem is efficiently solvable, and can be resolved by repeated applications of the *maximum flow problem* [13]. This is because the maximum flow between source s and sink t determines the minimum s-t cut. By using different source and sink combinations, it is also possible to find the global minimum cut. A second way of determining a minimum cut is by using a contraction-based edge-sampling approach. This is a probabilistic technique in which we successively sample edges in order to collapse nodes into larger sets of nodes. By successively sampling different sequences of edges and picking the optimum value [177], it is possible to determine a global minimum cut. Both of the above techniques are quite efficient and the time-complexity is polynomial in terms of the number of nodes and edges. An interesting discussion of this problem may be found in [78].

The *multi-way graph partitioning problem* is significantly more difficult, and is NP-hard [80]. In this case, we wish to partition a graph into $k > 2$ components, so that the total weight of the edges whose ends lie in different partitions is minimized. A well known technique for graph partitioning is the Kerninghan-Lin algorithm [116]. This classical algorithm is based on a hill-

climbing (or more generally neighborhood-search technique) for determining the optimal graph partitioning. Initially, we start off with a random cut of the graph. In each iteration, we exchange a pair of vertices in two partitions, to see if the overall cut value is reduced. In the event that the cut value is reduced, then the interchange is performed. Otherwise, we pick another pair of vertices in order to perform the interchange. This process is repeated until we converge to a optimal solution. We note that this optimum may not be a global optimum, but may only be a local optimum of the underlying data. The main variation in different versions of the Kerninghan-Lin algorithm is the policy which is used for performing the interchanges on the vertices. We note that the use of more sophisticated strategies allows a better improvement in the objective function for each interchange, but also requires more time for each interchange. This is a natural tradeoff which may work out differently depending upon the nature of the application at hand. We note that the problem of graph partitioning is studied widely in the literature. A detailed survey may be found in [77].

A closely related problem is that of dense subgraph determination in massive graphs. This problem is frequently encountered in large graph data sets. For example, the problem of determining large subgraphs of web graphs was studied in [82]. In this paper, a min-hash approach was used to determine the *shingles* which represent dense subgraphs. The broad idea is to represent the outlinks of a particular node as sets. Two nodes are considered similar, if they share many outlinks. Thus, consider a node A with an outlink set S_A and a node B with outlink set S_B. Then the similarity between the two nodes is defined by the *Jaccard coefficient*, which is defined as $\frac{S_A \cap S_B}{S_A \cup S_B}$. We note that explicit enumeration of all the edges in order to compute this can be computationally inefficient. Rather, a *min-hash approach* is used in order to perform the estimation. This *min-hash approach* is as follows. We sort the universe of nodes in a random order. For any set of nodes in random sorted order, we determine the first node $First(A)$ for which an outlink exists from A to $First(A)$. We also determine the first node $First(B)$ for which an outlink exists from B to $First(B)$. It can be shown that the Jaccard coefficient is an unbiased estimate of the probability that $First(A)$ and $First(B)$ are the same node. By repeating this process over different permutations over the universe of nodes, it is possible to accurately estimate the Jaccard Coefficient. This is done by using a constant number of permutations c of the node order. Thus, for each node, a fingerprint of size c can be constructed. By comparing the fingerprints of two nodes, the Jaccard coefficient can be estimated. This approach can be further generalized with the use of every s element set contained entirely with S_A and S_B. By using different values of s and c, it is possible to design an algorithm which distinguishes between two sets that are above or below a certain threshold of similarity.

The overall technique in [82] first generates a set of c shingles of size s for each node. The process of generating the c shingles is extremely straightforward. Each node is processed independently. We use the min-wise hash function approach in order to generate subsets of size s from the outlinks at each node. This results in c subsets for each node. Thus, for each node, we have a set of c shingles. Thus, if the graph contains a total of n nodes, the total size of this shingle fingerprint is $n \times c \times sp$, where sp is the space required for each shingle. Typically sp will be $O(s)$, since each shingle contains s nodes. For each distinct shingle thus created, we can create a list of nodes which contain it. In general, we would like to determine groups of shingles which contain a large number of common nodes. In order to do so, the method in [82] performs a second-order shingling in which the meta-shingles are created from the shingles. Thus, this further compresses the graph in a data structure of size $c \times c$. This is essentially a constant size data structure. We note that this group of meta-shingles has the the property that they contain a large number of common nodes. The dense subgraphs can then be extracted from these meta-shingles. More details on this approach may be found in [82].

A related problem is that of determining quasi-cliques in the underlying data. Quasi-cliques are essentially relaxations on the concept of cliques. In the case of a clique, the subgraph induced on a set of nodes is *complete*. On the other hand, in the case of a γ-quasi-clique, each vertex in that subset of nodes has a degree of at least $\gamma \cdot k$, where γ is a fraction, and k is the number of nodes in that set. The first work on determining γ-quasi-cliques was discussed in [5], in which a randomized algorithm is used in order to determine a quasi-clique with the largest size. A closely related problem is that of finding *frequently occurring cliques* in *multiple data sets*. In other words, when multiple graphs are obtained from different data sets, some dense subgraphs occur frequently together in the different data sets. Such graphs help in determining *important dense patterns of behavior in different data sources*. Such techniques find applicability in mining important patterns in graphical representations of customers. The techniques are also helpful in mining cross-graph quasi-cliques in gene expression data. A description of the application of the technique to the problem of gene-expression data may be found in [153]. An efficient algorithm for determining cross graph quasi-cliques was proposed in [148].

Classical Algorithms for Clustering XML and Graph Data. In this section, we will discuss a variety of algorithms for clustering XML and graph data. We note that XML data is quite similar to graph data in terms of how the data is organized structurally. In has been shown in [8, 63, 126, 133] that the use of this structural behavior is more critical for effective processing. There are two main techniques used for clustering of XML documents. These techniques are as follows:

- **Structural Distance-based Approach:** This approach computes structural distances between documents and uses them in order to compute clusters of documents. Such distance-based approaches are quite general and effective techniques over a wide variety of non-numerical domains such as categorical and string data. It is therefore natural to explore this technique in the context of graph data. One of the earliest work on clustering tree structured data is the *XClust algorithm* [126], which was designed to cluster XML schemas for efficient integration of large numbers of Document Type Definitions (DTDs) of XML sources. It adopts the agglomerative hierarchical clustering method which starts with clusters of single DTDs and gradually merges the two most similar clusters into one larger cluster. The similarity between two DTDs is based on their element similarity, which can be computed according to the semantics, structure, and context information of the elements in the corresponding DTDs. One of the shortcomings of the XClust algorithm is that it does not make full use of the structure information of the DTDs, which is quite important in the context of clustering tree-like structures. The method in [45] computes similarity measures based on the structural edit-distance between documents. This edit-distance is used in order to compute the distances between clusters of documents.

  Another clustering technique which falls in this general class of methods is the *S-GRACE* algorithm. The main idea is to use the element-subelement relationships in the distance function rather than the simple use of the tree-edit distance as in [45]. S-GRACE is a hierarchical clustering algorithm [133]. In [133], an XML document is converted to a structure graph (or s-graph), and the distance between two XML documents is defined according to the number of the common element-subelement relationships, which can capture better structural similarity relationships than the tree edit distance in some cases [133].

- **Structural Summary Based Approach:** In many cases, it is possible to create summaries from the underlying documents. These summaries are used for creating groups of documents which are similar to these summaries. The first summary-based approach for clustering XML documents was presented in [63]. In [63], the XML documents are modeled as rooted ordered labeled trees. A framework for clustering XML documents by using structural summaries of trees is presented. The aim is to improve algorithmic efficiency without compromising cluster quality.

  A second approach for clustering XML documents is presented in [8], and is referred to as *XProj*. This technique is a partition-based algorithm. The primary idea in this approach is to use frequent-pattern mining algorithms in order to determine the summaries of frequent structures in the

data. The technique uses a k-means type approach in which each cluster center comprises a set of frequent patterns which are local to the partition for that cluster. The frequent patterns are mined using the documents assigned to a cluster center in the last iteration. The documents are then further re-assigned to a cluster center based on the average similarity between the document and the newly created cluster centers from the local frequent patterns. In each iteration the document-assignment and the mined frequent patterns are iteratively re-assigned, until the cluster centers and document partitions converge to a final state. It has been shown in [8] that such a structural summary based approach is significantly superior to a similarity function based approach as presented in [45]. The method is also superior to the structural approach in [63] because of its use of more robust representations of the underlying structural summaries.

3.3 Classification Algorithms for Graph Data

Classification is a central task in data mining and machine learning. As graphs are used to represent entities and their relationships in an increasing variety of applications, the topic of graph classification has attracted much attention in both academia and industry. For example, in pharmaceutics and drug design, we are interested to know the relationship between the activity of a chemical compound and the structure of the compound, which is represented by a graph. In social network analysis, we study the relationship between the health of a community (e.g., whether it is expanding or shrinking) and its structure, which again is represented by graphs.

Graph classification is concerned with two different but related learning tasks.

- **Label Propagation.** A subset of nodes in a graph are labeled. The task is to learn a model from the labeled nodes and use the model to classify the unlabeled nodes.
- **Graph classification.** A subset of graphs in a graph dataset are labeled. The task is to learn a model from the labeled graphs and use the model to classify the unlabeled graphs.

Label Propagation. The concept of *label or belief propagation* [174, 209, 210] is a fundamental technique which is used in order to leverage graph structure in the context of classification in a number of relational domains. The scenario of label propagation [44] occurs in many applications. As an example, social network analysis is being used as a mean for targeted marketing. Retailers track customers who have received promotions from them. Those customers who respond to the promotion (by making a purchase) are labeled

as positive nodes in the graph representing the social network, and those who do not respond are labeled as negative. The goal of target marketing is to send promotions to customers who are most likely to respond to promotions. It boils down to learning a model from customers who have received promotions and predicting the responses of other potential customers in the social network. Intuitively, we want to find out how existing positive and negative labels propagate in the graph to unlabeled nodes.

Based on the assumption that "similar" nodes should have similar labels, the core challenge for label propagation lies in devising a distance function that measures the similarity between two nodes in the graph. One common approach of defining the distance between two nodes is to count the average number of steps it takes to reach one node from the other using a random walk [119, 178]. However, it has a significant drawback: it takes $O(n^3)$ time to derive the distances and $O(n^2)$ space to store the distances between all pairs. However, many graphs in real life applications are sparse, which reduces the complexity of computing the distance [211, 210]. For example, Zhou et al [210] introduces a method whose complexity is nearly linear to the number of non-zero entries of the sparse coefficient matrix. A survey of label propagation methods can be found in [179].

Kernel-based Graph Classification Methods. Kernel-based graph classification employs a graph kernel to measure the similarity between two labeled graphs. The method is based on random walks. For each graph, we enumerate its paths, and we derive probabilities for such paths. The graph kernel compares the set of paths and their probabilities between the two graphs. A random path (represented as a sequence of node and edge labels) is generated via a random walk: First, we randomly select a node from the graph. During the next and each of the subsequent steps, we either stop (the path ends) or randomly select an adjacent node to continue the random walk. The choices we make are subject to a given stopping probability and a node transition probability. By repeating the random walks, we derive a table of paths, each of which is associated with a probability.

In order to measure the similarity between two graphs, we need to measure the similarity between nodes, edges, and paths.

- **Node/Edge kernel.** An example of a node/edge kernel is the identity kernel. If two nodes/edges have the same label, then the kernel returns 1 otherwise 0. If the node/edge labels take real values, then a Gaussian kernel can be used instead.

- **Path kernel.** A path is a sequence of node and edge labels. If two paths are of the same length, the path kernel can be constructed as the product

of node and edge kernels. If two paths are of different lengths, the path kernel simply returns 0.

- **Graph kernel.** As each path is associated with a probability, we can define the graph kernel as the expectation of the path kernel over all possible paths in the two graphs.

The above definition of a graph kernel is straightforward. However, it is computationally infeasible to enumerate all the paths. In particular, in cyclic graphs, the length of a path is unbounded, which makes enumeration impossible. Thus, more efficient approaches are needed to compute the kernel. It turns out that the definition of the kernel can be reformulated to show a nested structure. In the case of directed acyclic graphs the nodes can be topologically ordered such that there is no path from node j to i if $i < j$, the kernel can be redefined as a recursive function, and dynamic programming can handle this problem in $O(|\mathcal{X}| \cdot |\mathcal{X}'|)$, where $\mathcal{X}$ and $\mathcal{X}'$ are the set of nodes in the two graphs. In the case of cyclic graphs, the kernel's feature space (label sequences) is possibly infinite because of loops. The computation of cyclic graph kernel can still be done with linear system theory and convergence properties of the kernel.

Boosting-based Graph Classification Methods. While the kernel-based method provides an elegant solution to graph classification, it does not explicitly reveal what graph features (substructures) are relevant for classification. To address this issue, a new approach of graph classification based on pattern mining is introduced. The idea is to perform graph classification based on a graph's important substructures. We can create a binary feature vector based on the presence or absence of a certain substructure (subgraph) and apply an off-the-shelf classifier.

Since the entire set of subgraphs is often very large, we must focus on a small subset of features that are relevant. The most straightforward approach for finding interesting features is through frequent pattern mining. However, frequent patterns are not necessarily relevant patterns. For instance, in chemical graphs, ubiquitous patterns such as C-C or C-C-C are frequent, but have almost no significance in predicting important characteristics of chemical compounds such as activity, toxicity, etc.

Boosting is used to automatically select a relevant set of subgraphs as features for classification. LPBoost (Linear Program Boost) learns a linear discriminant function for feature selection. To obtain an interpretable rule, we need to obtain a sparse weight vector, where only a few weights are nonzero. It was shown [162] that graph boosting can achieve better accuracy than graph kernels, and it has the advantage of discovering key substructures explicitly at the same time.

The problem of graph classification is closely related to that of XML classification. This is because XML data can be considered an instance of *rich graphs*, in which nodes and edges have features associated with them. Consequently, many of the methods for XML classification can also be used for structural graph classification. In [205], a rule-based classifier (called *XRules*) was proposed in which we associate structural features on the left-hand side with class labels on the right-hand side. The structural features on the left-hand side are determined by computing the structural features in the graph which are both *frequent* and *discriminative* for classification purposes. These structural features are used in order to construct a prioritized list of rules which are used for classification purposes. The top-k rules are determined based on the discriminative behavior and the majority class label on the right hand side of these k rules is reported as the final result.

Other Related Work. The problem of node classification arises in a number of different application contexts such as relational data classification, social network classification, and blog classification. A technique has been proposed in [138], which uses link-based similarity for node-classification in the context of relational data. This approach constructs *link features* from the underlying structure and uses them in order to create an effective model for classification. Recently, this technique has also been used in the context of link-based classification of blogs [23]. However, all of these techniques use link-based methods only. Since many of these techniques arise in the context of text data, it is natural to examine whether such content can be used in order to improve classification accuracy. A method to perform *collective classification* of email speech acts has been proposed in [39]. It has been shown that the analysis of relational aspects of emails (such as emails in a particular thread) significantly improves the classification accuracy. It has also been shown in [206] that the use of graph structures during categorization improves the classification accuracy of web pages. Another work [25] discusses the problem of label acquisition in the context of collective classification.

3.4 The Dynamics of Time-Evolving Graphs

Many networks in real applications arise in the context of networked entities such as the web, mobile networks, military networks, and social networks. In such cases, it is useful to examine various aspects of the *evolution dynamics* of *typical networks*, such as the web or social networks. Thus, this line of research focusses on modeling the general evolution properties of very large graphs which are *typically* encountered. Considerable study has been devoted to that of examining generic evolution properties which *hold across massive networks such as web networks, citation networks and social networks.* Some examples of such properties are as follows:

Densification: Most real networks such as the web and social networks continue to become more dense over time [129]. This essentially means that these networks continue to add more links over time (than are deleted). This is a natural consequence of the fact that much of the web and social media is a relatively recent phenomenon for which new applications continue to be found over time. In fact most real graphs are known to exhibit a *densification power law*, which characterizes the variation in densification behavior over time. This law states that the number of nodes in the network increases superlinearly with the number of nodes over time, whereas the number of edges increases superlinearly over time. In other words, if $n(t)$ and $e(t)$ represent the number of edges and nodes in the network at time t, then we have:

$$e(t) \propto n(t)^{\alpha} \tag{2.1}$$

The value of the exponent α lies between 1 and 2.

Shrinking Diameters: The *small world* phenomenon of graphs is well known. For example, it was shown in [130] that the average path length between two MSN messenger users is 6.6. This can be considered a verification of the (internet version of the) widely known rule of "six degrees of separation" in (generic) social networks. It was further shown in [129], that the diameters of massive networks such as the web continue to shrink over time. This may seem surprising, because one would expect that the diameter of the network should grow as more nodes are added. However, it is important to remember that edges are added more rapidly to the network than nodes (as suggested by Equation 2.1 above). As more edges are added to the graph it becomes possible to traverse from one node to another with the use of a fewer number of edges.

While the above observations provide an understanding of some key aspects of specific aspects of long-term evolution of massive graphs, they do not provide an idea of how the evolution in social networks can be *modeled* in a comprehensive way. A method which was proposed in [131] uses the *maximum likelihood principle* in order to characterize the evolution behavior of massive social networks. This work uses data-driven strategies in order to model the online behavior of networks. The work studies the behavior of four different networks, and uses the observations from these networks in order to create a model of the underlying evolution. It also shows that edge locality plays an important role in the evolution of social networks. A complete model of a node's behavior during its lifetime in the network is studied in this work.

Another possible line of work in this domain is to study methods for characterizing the evolution of specific graphs. For example, in a social network, it may be useful to determine the newly forming or decaying communities in the underlying network [9, 16, 50, 69, 74, 117, 131, 135, 171, 173]. It was shown in [9] how expanding or contracting communities in a social network may be characterized by examining the relative behavior of edges, as they are received

in a dynamic graph stream. The techniques in this paper characterize the structural behavior of the incremental graph within a given time window, and uses it in order to determine the birth and death of communities in the graph stream. This is the first piece of work which studies the problem of evolution in *fast streams of graphs*. It is particularly challenging to study the stream case, because of the inherent combinatorial complexity of graph structural analysis, which does not lend itself well to the stream scenario.

The work in [69] uses statistical analysis and visualization in order to provide a better idea of the changing community structure in an evolving social network. A method in [171] performs parameter-free mining of large time-evolving graphs. This technique can determine the evolving communities in the network, as well as the critical change-points in time. A key property of this method is that it is *parameter-free*, and this increases the usability of the method in many scenarios. This is achieved with the use of the MDL principle in the mining process. A related technique can also perform parameter-free analysis of evolution in massive networks [74] with the use of the MDL principle. The method can determine which communities have shrunk, split, or emerged over time.

The problem of evolution in graphs is usually studied in the context of clustering, because clusters provide a natural summary for understanding both the underlying graph and the changes inherent during the evolution process. The need for such characterization arises in the context of massive networks, such as interaction graphs [16], community detection in social networks [9, 50, 135, 173], and generic clustering changes in linked information networks [117]. The work by [16] provides an *event based framework*, which provides an understanding of the typical events which occur in real networks, when new communities may form, evolve, or dissolve. Thus, this method can provide an easy way of making a quick determination of whether specific kinds of changes may be occurring in a particular network. A key technique used by many methods is to analyze the communities in the data over specific time slices and then determine the change between the slices to diagnose the nature of the underlying evolution. The method in [135] deviates from this two-step approach and constructs a unified framework for the determination of communities with the use of a best fit to a temporal-smoothness model. The work in [50] presents a spectral method for evolutionary clustering, which is also based on the temporal-smoothness concept. The method in [173] studies techniques for evolutionary characterization of networks in multi-modal graphs. Finally, a recent method proposed in [117] combines the problem of clustering and evolutionary analysis into one framework, and shows how to determine evolving clusters in a dynamic environment. The method in [117] uses a density-based characterization in order to construct *nano-clusters* which are further leveraged for evolution analysis.

A different approach is to use association rule-based mining techniques [22]. The algorithm takes a sequence of snapshots of an evolving graph, and then attempts to determine rules which define the changes in the underlying graph. Frequently occurring sequences of changes in the underlying graph are considered important indicators for rule determination. Furthermore, the frequent patterns are decomposed in order to study the confidence that a particular sequence of steps in the past will lead to a particular transition. The probability of such a transition is referred to as *confidence*. The rules in the underlying graph are then used in order to characterize the overall network evolution.

Another form of evolution in the networks is in terms of the underlying *flow of communication (or information)*. Since the flow of communication and information implicitly defines a graph (stream), the dynamics of this behavior can be very interesting to study for a number of different applications. Such behaviors arise often in a variety of information networks such as social networks, blogs, or author citation graphs. In many cases, the evolution may take the form of cascading information through the underlying graphs. The idea is that information propagates through the social network through contact between the different entities in the network. The evolution of this information flow shares a number of similarities with the spread of diseases in networks. We will discuss more on this issue in a later section of this paper. Such evolution has been studied in [128], which studies how to characterize the evolution behavior in blog graphs.

4. Graph Applications

In this section, we will study the application of many of the aforementioned mining algorithms to a variety of graph applications. Many data domains such as chemical data, biological data, and the web are naturally structured as graphs. Therefore, it is natural that many of the mining applications discussed earlier can be leveraged for these applications. In this section, we will study the diverse applications that graph mining techniques can support. We will also see that even though these applications are drawn from different domains, there are some common threads which can be leveraged in order to improve the quality of the underlying results.

4.1 Chemical and Biological Applications

Drug discovery is a time consuming and extremely expensive undertaking. Graphs are natural representations for chemical compounds. In chemical graphs, nodes represent atoms and edges represent bonds between atoms. Biology graphs are usually on a higher level where nodes represent amino acids and edges represent connections or contacts among amino acids. An important assumption, which is known as the structure activity relationship (SAR) princi-

ple, is that the properties and biological activities of a chemical compound are related to its structure. Thus, graph mining may help reveal chemical and biology characteristics such as activity, toxicity, absorption, metabolism, etc. [30], and facilitate the process of drug design. For this reason, academia and pharmaceutical industry have stepped up efforts in chemical and biology graph mining, in the hope that it will dramatically reduce the time and cost in drug discovery.

Although graphs are natural representations for chemical and biology structures, we still need a computationally efficient representation, known as descriptors, that is conducive to operations ranging from similarity search to various structure driven predictions. Quite a few descriptors have been proposed. For example, hash fingerprints [2, 1] are a vectorized representation. Given a chemical graph, we create a a hash fingerprint by enumerating certain types of basic structures (e.g., cycles and paths) in the graph, and hashing them into a bit-string. In another line of work, researchers use data mining methods to find frequent subgraphs [150] in a chemical graph database, and represent each chemical graph as a vector in the feature space created by the set of frequent subgraphs. A detailed description and comparison of various descriptors can be found in [190].

One of the most fundamental operations on chemical compounds is similarity search. Various graph matching algorithms have been employed for i) *rank-retrieval*, that is, searching a large database to find chemical compounds that share the same bioactivity as a query compound; and ii) *scaffold-hopping*, that is, finding compounds that have similar bioactivity but different structure from the query compound. Scaffold-hopping is used to identify compounds that are good "replacement" for the query compound, which either has some undesirable properties (e.g., toxicity), or is from the existing patented chemical space. Since chemical structure determines bioactivity (the SAR principle), scaffold-hopping is challenging, as the identified compounds must be structurally similar enough to demonstrate similar bioactivity, but different enough to be a novel chemotype. Current approaches for similarity matching can be classified into two categories. One category of approaches perform similarity matching directly on the descriptor space [192, 170, 207]. The other category of approaches also consider indirect matching: if a chemical compound c is structurally similar to the query compound q, and another chemical compound c' is structurally similar to c, then c' and q are indirect matches. Clearly, indirect macthing has the potential to indentify compounds that are functionally similar but structurally different, which is important to scaffold-hopping [189, 191].

Another important application area for chemical and biology graph mining is structure-driven prediction. The goal is to predict whether a chemical structure is active or inactive, or whether it has certain properties, for example, toxic or nontoxic, etc. SVM (Support Vector Machines) based methods have proved

effective for this task. Various vector space based kernel functions, including the widely used radial basis function and the Min-Max kernel [172, 192], are used to measure the similarity between chemical compounds that are represented by vectors. Instead of working on the vector space, another category of SVM methods use graph kernels to compare two chemical structures. For instance, in [160], the size of the maximum common subgraph of two graphs is used as a similarity measure.

In late 1980's, the pharmaceutical industry embraced a new drug discovery paradigm called target-based drug discovery. Its goal is to develop a drug that selectively modulates the effects of the disease-associated gene or gene product without affecting other genes or molecular mechanisms in the organism. This is made possible by the High Throughput Screening (HTS) technique, which is able to rapidly testing a large number of compounds based on their binding activity against a given target. However, instead of increasing the productivity of drug design, HTS slowed it down. One reason is that a large number of screened candidates may have unsatisfactory phenotypic effects such as toxity and promiscuity, which may dramatically increase the validation cost in later stage drug discovery [163]. Target Fishing [109] tackles the above issues by employing computational techniques to directly screen molecules for desirable phenotype effects. In [190], we offer a detailed description of various such methods, including multi-category Bayesian models [149], SVM rank [188], Cascade SVM [188, 84], and Ranking Perceptron [62, 188].

4.2 Web Applications

The world wide web is naturally structured in the form of a graph in which the web pages are the nodes and the links are the edges. The linkage structure of the web holds a wealth of information which can be exploited for a variety of data mining purposes. The most famous application which exploits the linkage structure of the web is the *PageRank* algorithm [29, 151]. This algorithm has been one of the key secrets to the success of the well known *Google* search engine. The basic idea behind the page rank algorithm is that the importance of a page on the web can be gauged from the number and importance of the hyperlinks pointing to it. The intuitive idea is to model a random surfer who follows the links on the pages with equal likelihood. Then, it is evident that the surfer will arrive more frequently at web pages which have a large number of paths leading to them. The intuitive interpretation of page rank is the probability that a random surfer arrives at a given web page during a random walk. Thus, the page rank essentially forms a probability distribution over web pages, so that the sum of the page rank over all the web pages sums to 1. In addition, we sometimes add teleportation, in which we can transition *any* web page in the collection uniformly at random.

Let A be the set of edges in the graph. Let π_i denote the steady state probability of node i in a random walk, and let $P = [p_{ij}]$ denote the transition matrix for the random-walk process. Let α denote the *teleportation probability* at a given step, and let q_i be the ith value of a probability vector defined over all the nodes which defines the probability that the teleportation takes place to node i at any given step (conditional on the fact that teleportation does take place). For the time-being, we assume that each value of q_i is the same, and is equal to $1/n$, where n is the total number of nodes. Then, for a given node i, we can derive the following steady-state relationship:

$$\pi_i = \sum_{j:(j,i)\in A} \pi_j \cdot p_{ji} \cdot (1-\alpha) + \alpha \cdot q_i \tag{2.2}$$

Note that we can derive such an equation for each node; this will result in a linear system of equations on the transition probabilities. The solutions to this system provides the page rank vector $\overline{\pi}$. This linear system has n variables, and n different constraints, and can therefore be expressed in n^2 space in the worst-case. The solution to such a linear systems requires matrix operations which are at least quadratic (and at most cubic) in the total number of nodes. This can be quite expensive in practice. Of course, since the page rank needs to be computed only once in a while in batch phase, it is possible to implement it reasonably well with the use of a few carefully designed matrix techniques. The *PageRank* algorithm [29, 151] uses an iterative approach which computes the principal eigenvectors of the normalized link matrix of the web. A description of the page rank algorithm may be found in [151].

We note that the page-rank algorithm only looks at the link structure during the ranking process, and does not include any information about the content of the underlying web pages. A closely related concept is that of *topic-sensitive page rank* [95], in which we use the topics of the web pages during the ranking process. The key idea in such methods is to allow for *personalized teleportation* (or jumps) during the random-walk process. At each step of the random walk, we allow a transition (with probability α) to a sample set S of pages which are related to the topic of the search. Otherwise, the random walk continues in its standard way with probability $(1-\alpha)$. This can be easily achieved by modifying the vector $\overline{q} = (q_1 \ldots q_n)$, so that we set the appropriate components in this vector to 1, and others to 0. The final steady-state probabilities with this modified random-walk defines the topic-sensitive page rank. The greater the probability α, the more the process biases the final ranking towards the sample set S. Since each topic-sensitive personalization vector requires the storage of a very large page rank vector, it is possible to pre-compute it in advance only in a limited way, with the use of some representative or authoritative pages. The idea is that we use a limited number of such personalization vectors $\overline{q}$ and determine the corresponding *personalized* page rank vectors $\overline{\pi}$

for these authoritative pages. A judicious combination of these different personalized page rank vectors (for the authoritative pages) is used in order to define the response for a given query set. Some examples of such approaches are discussed in [95, 108]. Of course, such an approach has limitations in terms of the level of granularity in which it can perform personalization. It has been shown in [79] that fully personalized page rank, in which we can precisely bias the random walk towards an *arbitrary* set of web pages will always require at least quadratic space in the worst-case. Therefore, the approach in [79] observes that the use of Monte-Carlo sampling can greatly reduce the space requirements without sufficiently affecting quality. The work in [79] pre-stores Monte-Carlo samples of node-specific random walks, which are also referred to as *fingerprints*. It has been shown in [79] that a very high level of accuracy can be achieved in limited space with the use of such fingerprints. Subsequent recent work [42, 87, 175, 21] has built on this idea in a variety of scenarios, and shown how such dynamic personalized page rank techniques can be made even more efficient and effective. Detailed surveys on different techniques for page rank computation may be found in [20].

Other relevant approaches include the use of measures such as the *hitting time* in order to determine and rank the context sensitive proximity of nodes. The hitting time between node i to j is defined as the expected number of hops that a random surfer would require to reach node j from node i. Clearly, the hitting time is a function of not just the length of the shortest paths, but also the number of possible paths which exist from node i to node j. Therefore, in order to determine similarity among linked objects, the hitting time is a much better measurement of proximity as compared to the use of shortest-path distances. A truncated version of the hitting time defines the objective function by restricting only to the instances in which the hitting time is below a given threshold. When the hitting time is larger than a given threshold, the contribution is simply set at the threshold value. Fast algorithms for computing a truncated variant of the hitting time are discussed in [164]. The issue of scalability in random-walk algorithms is critical because such graphs are large and dynamic, and we would like to have the ability to rank quickly for particular kinds of queries. A method in [165] proposes a fast dynamic re-ranking method, when user feedback is incorporated. A related problem is that of investigating the behavior of random walks of fixed length. The work in [203] investigates the problem of neighborhood aggregation queries. The aggregation query can be considered an "inverse version" of the hitting time, where we are fixing the number of hops and attempting to determine the *number* of hits, rather than the number of hops to hit. One advantage of this definition is that it automatically considers only truncated random walks in which the length of the walk is below a given threshold h; it is also a cleaner definition than the truncated hitting time by treating different walks in a uniform way. The work in [203] determines nodes

that have the top-k highest aggregate values over their h-hop neighbors with the use of a Local Neighborhood Aggregation framework called LONA. The framework exploits locality properties in network space to create an efficient index for this query.

Another related idea on determining authoritative ranking is that of the *hub-authority model* [118]. The page-rank technique determines authority by using linkage behavior as indicative of authority. The work in [118] proposes that web pages are one of two kinds:

- **Hubs** are pages which link to authoritative pages.
- **Authorities** are pages which are linked to by good hubs.

A score is associated with both hubs and authorities corresponding to their goodness for being hubs and authorities respectively. The hubs scores affect the authority scores and vice-versa. An iterative approach is used in order to compute both the hub and authority scores. The HITS algorithm proposed in [118] uses these two scores in order to compute the hubs and authorities in the web graph.

Many of these applications arise in the context of dynamic graphs in which the nodes and edges of the graph are received over time. For example, in the context of a social network in which new links are being continuously created, the estimation of page rank is inherently a dynamic problem. Since the page rank algorithm is critically dependent upon the behavior of random walks, the streaming page rank algorithm [166] samples nodes independently in order to create short random walks from each node. This walks can then be merged to create longer random walks. By running several such random walks, the page rank can be effectively estimated. This is because the page rank is simply the probability of visiting a node in a random walk, and the sampling algorithm simulates this process well. The key challenge for the algorithm is that it is possible to get stuck during the process of random walks. This is because the sampling process picks both nodes and edges in the sample, and it is possible to traverse an edge such that the end point of that edge is not present in the node sample. Furthermore, we do not allow repeated traversal of nodes in order to preserve randomness. Such stuck nodes can be handled by keeping track of the set S of sampled nodes whose walks have already been used for extending the random walk. New edges are sampled out of both the stuck node and the nodes in S. These are used in order to extend the walk further as much as possible. If the new end-point is a sampled node whose walk is not in S, then we continue the merging process. Otherwise, we repeat the process of sampling edges out of S and all the stuck nodes visited since the last walk was used.

Another application commonly encountered in the context of graph mining is the analysis of query flow logs. We note that a common way for many users to navigate on the web is to use search engines to discover web pages and then

click some of the hyperlinks in the search results. The behavior of the resulting graphs can be used to determine the topic distributions of interest, and semantic relationships between different topics.

In many web applications, it is useful to determine clusters of web pages or blogs. For this purpose, it is helpful to leverage the linkage structure of the web. A common technique which is often used for web document clustering is that of *shingling* [32, 82]. In this case, the min-hash approach is used in order to determine densely connected regions of the web. In addition, any of a number of quasi-clique generation techniques [5, 148, 153] can be used for the purpose of determination of dense regions of the graph.

Social Networking. Social networks are very large graphs which are defined by people who appear as nodes, and links which correspond to communications or relationships between these different people. The links in the social network can be used to determine relevant communities, members with particular expertise sets, and the flow of information in the social network. We will discuss these applications one by one.

The problem of community detection in social networks is related to the problem of *node clustering* of very large graphs. In this case, we wish to determine dense clusters of nodes based on the underlying linkage structure [158]. Social networks are a specially challenging case for the clustering problem because of the typically massive size of the underlying graph. As in the case of web graphs, any of the well known shingling or quasi-clique generation methods [5, 32, 82, 148, 153] can be used in order to determine relevant communities in the network. A technique has been proposed in [167] to use stochastic flow simulations for determining the clusters in the underlying graphs. A method for determining the clustering structure with the use of the eigen-structure of the linkage matrix in order to determine the community structure is proposed in [146]. An important characteristic of large networks is that they can often be characterized by the nature of the underlying subgraphs. In [27], a technique has been proposed for counting the number of subgraphs of a particular type in a large network. It has been shown that this characterization is very useful for clustering large networks. Such precision cannot be achieved with the use of other topological properties. Therefore, this approach can also be used for community detection in massive networks. The problem of community detection is particularly interesting in the context of *dynamic analysis* of evolving networks in which we try to determine how the communities in the graph may change over time. For example, we may wish to determine *newly forming communities*, *decaying communities*, or *evolving communities*. Some recent methods for such problems may be found in [9, 16, 50, 69, 74, 117, 131, 135, 171, 173]. The work in [9] also examines this problem in the context of evolving graph streams. Many of these techniques

examine the problem of community detection and change detection in a single framework. This provides the ability to present the changes in the underlying network in a summarized way.

Node clustering algorithms are closely related to the concept of *centrality analysis* in networks. For example, the technique discussed in [158] uses a k-medoids approach which yields k central points of the network. This kind of approach is very useful in different kinds of networks, though in different contexts. In the case of social networks, these central points are typically key members in the network which are well connected to other members of the community. Centrality analysis can also be used in order to determine the central points in information flows. Thus, it is clear that the same kind of structural analysis algorithm can lead to different kinds of insights in different networks.

Centrality detection is closely related to the problem of information flow spread in social networks. It was observed that many recently developed viral flow analysis techniques [40, 127, 147] can be used in the context of a variety of other social networking information flow related applications. This is because information flow applications can be understood with similar behavior models as viral spread. These applications are: (1) We would like to determine the most influential members of the social network; i.e. members who cause the most flow of information outwards. (2) Information in the social behavior often cascades through it in the same way as an epidemic. We would like to measure the information cascade rate through the social network, and determine the effect of different sources of information. The idea is that monitoring promotes the early detection of information flows, and is beneficial to the person who can detect it. The cascading behavior is particularly visible in the case of blog graphs, in which the cascading behavior is reflected in the form of added links over time. Since it is not possible to monitor all blogs simultaneously, it is desirable to minimize the monitoring cost over the different blogs, by assuming a fixed monitoring cost per node. This problem is NP-hard [127], since the vertex-cover problem can be reduced to it. The main idea in [128] is to use an approximation heuristic in order to minimize the monitoring cost. Such an approach is not restricted to the blog scenario, but it is also applicable to other scenarios such as monitoring information exchange in social networks, and monitoring outages in communication networks. (3) We would like to determine the conditions which lead to the critical mass necessary for uncontrolled information transmission. Some techniques for characterizing these conditions are discussed in [40, 187]. The work in [187] relates the structure of the adjacency matrix to the transmissibility rate in order to measure the threshold for an epidemic. Thus, the connectivity structure of the underlying graph is critical in measuring the rate of information dissemination in the underlying

network. It has been shown in [187] that the eigenstructure of the adjacency matrix can be directly related to the threshold for an epidemic.

Other Computer Network Applications. Many of these techniques can also be used for other kinds of networks such as communication networks. Structural analysis and robustness of communication networks is highly dependent upon the design of the underlying network graph. Careful design of the underlying graph can help avoid network failures, congestions, or other weaknesses in the overall network. For example, centrality analysis [158] can be used in the context of a communication network in order to determine critical points of failure. Similarly, the techniques for flow dissemination in social networks can be used to model viral transmission in communication networks as well. The main difference is that we model viral infection probability along an edge in a communication network instead of the information flow probability along an edge in a social network.

Many reachability techniques [10, 48, 49, 53, 54, 184] can be used to determine optimal routing decisions in computer networks. This is also related to the problem of determining pairwise node-connectivity [7] in computer networks. The technique in [7] uses a compression-based synopsis to create an effective connectivity index for massive disk-resident graphs. This is useful in communication networks in which we need to determine the minimum number of edges to be deleted in order to disconnect a particular pair of nodes from one another.

4.3 Software Bug Localization

A natural application of graph mining algorithms is that of software bug localization. Software bug localization is an important application from the perspective of software reliability and testing. The control flow of programs can be modeled in the form of call-graphs. The goal of software bug localization techniques is to mine such call graphs in order to determine the bugs in the underlying programs. Call graphs are of two types:

- **Static call graphs** can be inferred from the source code of a given program. All the methods, procedures and functions in the program are nodes, and the relationships between the different methods are defined as edges. It is also possible to define nodes for data elements and model relationships between different data elements and edges. In the case of static call graphs, it is often possible to use *typical examples* of the structure of the program in order to determine portions of the software where atypical anamolies may occur.

- **Dynamic call graphs** are created during program execution, and they represent the invocation structure. For example, a call from one pro-

cedure to another creates an edge which represents the invocation relationship between the two procedures. Such call graphs can be extremely large in massive software programs, since such programs may contain thousands of invocations between the different procedures. In such cases, the difference in structural, frequency or sequence behavior of successful and failing invocations can be used to localize software bugs. Such call graphs can be particularly useful in localizing bugs which are occasional in nature and may occur in some invocations and not others.

We further note that bug localization is not exhaustive in terms of the kinds of errors it can catch. For example, logical errors in a program which are not a result of the program structure, and which do not affect the sequence or structure of execution of the different methods cannot be localized with such techniques. Furthermore software bug localization is not an exact science. Rather, it can be used in order to provide software testing experts with possible bugs, and they can use this in order to make relevant corrections.

An interesting case is one in which different program executions lead to different structure, sequence and frequency of executions which are specific to failures and successes of the final program execution. These failures and successes may be a result of logical errors, which lead to changes in structure and frequency of method calls. In such cases, the software bug-localization can be modeled as a classification problem. The first step is to create call graphs from the executions. This is achieved by tracing the program executions during the testing process. We note that such call graphs may be huge and unwieldy for use with graph mining algorithms. The large sizes of call-graphs creates a challenge for graph mining procedures. This is because graph mining algorithms are often designed for relatively small graphs, whereas such call graphs may be huge. Therefore, a natural solution is to reduce the size of the call graph with the use of a compression based approach. This naturally results in loss of information, and in some cases, it also results in an inability to use the localization approach effectively when the loss of information is extensive.

The next step is to use frequent subgraph mining techniques on the training data in order to determine those patterns which occur more frequently in faulty executions. We note that this is somewhat similar to the technique often utilized in rule-based classifiers which attempt to link particular patterns and conditions to specific class labels. Such patterns are then associated with the different methods and are used in order to provide a ranking of the methods and functions in the program which may possibly contain bugs. This also provides a causality and understanding of the bugs in the underlying programs.

We note that the compression process is critical in providing the ability to efficiently process the underlying graphs. One natural method for reducing the size of the corresponding graphs is to map multiple nodes in the call graph

into a single node. For example, in *total reduction*, we map every node in the call node which corresponds to the same method onto one node in the compressed graph. Thus, the total number of nodes in the graph is at most equal to the number of methods. Such a technique has been used in [136] in order to reduce the size of the call graph. A second method which may be used is to compress the iteratively executed structures such as loops into a single node. This is a natural approach, since an iteratively executed structure is one of the most commonly occurring blocks in call graphs. Another technique is to reduce subtrees into single nodes. A variety of localization strategies with the use of such reduction techniques are discussed in [67, 68, 72].

Finally, the reduced graphs are mined in order to determine discriminative structures for bug localization. The method in [72] is based on determining discriminative subtrees from the data. Specifically, the method finds all subtrees which are frequent to failing executions, but are not frequent in correct executions. These are then used in order to construct rules which may be used for specific instances of classification of program runs. More importantly, such rules provide an understanding of the causality of the bugs, and this understanding can be used in order to support the correction of the underlying errors.

The above technique is designed for finding structural characteristics of the execution which can be used for isolating software bugs. However, in many cases the structural characteristics may not be the only features which may be relevant to localization of bugs. For example, an important feature which may be used in order to determine the presence of bugs is the *relative frequency* of the invocation of different methods. For example, invocations which have bugs may call a particular method more frequently than others. A natural way to learn this is to associate edge weights with the call graph. These edge weights correspond to the frequency of invocation. Then, we use these edge weights in order to analyze the calls which are most relevant to discriminating between correct and failing executions. A number of methods for this class of techniques is discussed in [67, 68].

We note that both structure and frequency are different aspects of the data which can be leveraged in order to perform the localization. Therefore, it makes sense to combine these approaches in order to improve the localization process. The techniques in [67, 68] create a score for both the structure-based and frequency-based features. A combination of these scores is then used for the bug localization process. It has been shown [67, 68] that such an approach is more effective than the use of either of the two features.

Another important characteristic which can be explored in future work is to analyze the *sequence of program calls*, rather than simply analyzing the dynamic call structure or the frequency of calls of the different methods. Some initial work [64] in this direction shows that sequence mining encodes excellent information for bug localization even with the use of simple methods.

However, this technique does not use sophisticated graph mining techniques in order to further leverage this sequence information. Therefore, it can be a fruitful avenue for future research to incorporate sequential information into the graph mining techniques which are currently available.

Another line of analysis is the analysis of static source code rather than the dynamic call graphs. In such cases, it makes more sense to look particular classes of bugs, rather than try to isolate the source of the execution error. For example, neglected conditions in software programs [43] can create failing conditions. For example, a *case* statement in a software program with a missing condition is a commonly occurring bug. In such cases, it makes sense to design domain-specific techniques for localizing the bug. For this purpose, techniques based on *static* program-dependence graphs are used. These are distinguished from the dynamic call graphs discussed above, in the sense that the latter requires execution of the program to create the graphs, whereas in this case the graphs are constructed in a static fashion. Program dependence graphs essentially create a graphical representation of the relationships between the different methods and data elements of a program. Different kinds of edges are used to denote control and data dependencies. The first step is to determine conditional rules [43] in a program which illustrates the program dependencies which are frequently occurring in a project. Then we search for (static) instantiations within the project which violate these rules. In many cases, such instantiations could correspond to neglected conditions in the software program.

The field of software bug localization faces a number of key challenges. One of the main challenges is that the work in the field has mostly focussed on smaller software projects. Larger programs are a challenge, because the corresponding call graphs may be huge and the process of graph compression may lose too much information. While some of these challenges may be alleviated with the development of more efficient mining techniques for larger graphs, some advantages may also be obtained with the use of better representations at the *modeling level*. For example, the nodes in the graph can be represented at a coarser level of granularity at the modeling phase. Since the modeling process is done with a better level of understanding of the possibilities for the bugs (as compared to an automated compression process), it is assumed that such an approach would lose much less information for bug localization purposes. A second direction is to combine the graph-based techniques with other effective statistical techniques [137] in order to create more robust classifiers. In future research, it should be reasonable to expect that larger software projects can be analyzed only with the use of such combined techniques which can make use of different characteristics of the underlying data.

5. Conclusions and Future Research

In this chapter, we presented a survey of graph mining and management applications. We also provide a survey of the common applications which arise in the context of graph mining applications. Much of the work in recent years has focussed on small and memory-resident graphs. Much of the future challenges arise in the context of *very large disk-resident graphs*. Other important applications are designed in the context of *massive graphs streams*. Graph streams arise in the context of a number of applications such as social networking, in which the communications between large groups of users are captured in the form of a graph. Such applications are very challenging, since the entire data cannot be localized on disk for the purpose of structural analysis. Therefore, new techniques are required to summarize the structural behavior of graph streams, and use them for a variety of analytical scenarios. We expect that future research will focus on the large-scale and stream-based scenarios for graph mining.

Notes

1. FLWOR is an acronym for FOR-LET-WHERE-ORDER BY-RETURN.

References

[1] Chemaxon. *Screen, Chemaxon Inc.*, 2005.

[2] Daylight. *Daylight Toolkit, Daylight Inc, Mission Viejo, CA, USA*, 2008.

[3] Oracle Spatial Topology and Network Data Models 10g Release 1 (10.1) **URL:** *http://www.oracle.com/technology/products/spatial /pdf/10g_network_model_twp.pdf*

[4] Semantic Web Challenge. **URL:** *http://challenge.semanticweb.org/*

[5] J. Abello, M. G. Resende, S. Sudarsky, Massive quasi-clique detection. *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN) (Cancun, Mexico)*. 598-612, 2002.

[6] S. Abiteboul, P. Buneman, D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[7] C. Aggarwal, Y. Xie, P. Yu. GConnect: A Connectivity Index for Massive Disk-Resident Graphs, *VLDB Conference*, 2009.

[8] C. Aggarwal, N. Ta, J. Feng, J. Wang, M. J. Zaki. XProj: A Framework for Projected Structural Clustering of XML Documents, *KDD Conference*, 2007.

[9] C. Aggarwal, P. Yu. Online Analysis of Community Evolution in Data Streams. *SIAM Conference on Data Mining*, 2005.