# Chapter 17

# Mining Graph Data

*"Structure is more important than content in the transmission of information."*—Abbie Hoffman

## 17.1 Introduction

Graphs are ubiquitous in a wide variety of application domains such as bioinformatics, chemical, semi-structured, and biological data. Many important properties of graphs can be related to their structure in these domains. Graph mining algorithms can, therefore, be leveraged for analyzing various domain-specific properties of graphs. Most graphs, encountered in real applications, are one of the two types:

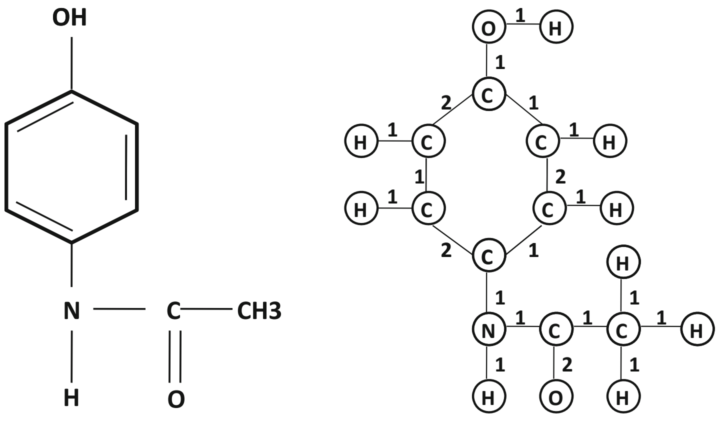
1. In applications such as chemical and biological data, a database of *many small graphs* is available. Each node is associated with a label that may or may not be unique to the node, depending on the application-specific scenario.

2. In applications such as the Web and social networks, a *single large graph* is available. For example, the Web can be viewed as a very large graph, in which nodes correspond to Web pages (labeled by their URLs) and edges correspond to hyperlinks between nodes.

The nature of the applications for these two types of data are quite different. Web and social network applications will be addressed in Chaps. 18 and 19, respectively. This chapter will therefore focus on the first scenario, in which many small graphs are available. A graph database may be formally defined as follows.

**Definition 17.1.1 (Graph Database)** *A graph database $\mathcal{D}$ is a collection of $n$ different undirected graphs, $G_1 = (N_1, A_1) \ldots G_n = (N_n, A_n)$, such that the set of nodes in the $i$th graph is denoted by $N_i$, and the set of edges in the $i$th graph is denoted by $A_i$. Each node $p \in N_i$ is associated with a label denoted by $l(p)$.*

The labels associated with the nodes may be repeated within a single graph. For example, when each graph $G_i$ corresponds to a chemical compound, the label of the node is the



(a) Acetaminophen (b) Graph representation

Figure 17.1: A chemical compound (*Acetaminophen*) and its associated graph representation

symbol denoting a chemical element. Because of the presence of multiple atoms of the same element, such a graph will contain label repetitions. The repetition of labels within a single graph leads to numerous challenges in graph matching and distance computation.

Graph data are encountered in many real applications. Some examples of key applications of graph data mining are as follows:

- Chemical and biological data can be expressed as graphs in which each node corresponds to an atom and a bond between a pair of atoms is represented by an edge. The edges may be weighted to reflect bond strength. An example of a chemical compound and its corresponding graph are illustrated in Fig. 17.1. Figure 17.1a shows an illustration of the chemical *acetaminophen*, a well-known analgesic. The corresponding graph representation is illustrated in Fig. 17.1b along with node labels and edge weights. In many graph mining applications, unit edge weights are assumed as a simplification.

- XML data can be expressed as attributed graphs. The relationships between different attributes of a structured record can be expressed as edges.

- Virtually any data type can be expressed as an entity-relationship graph. This provides a different way of mining conventional database records when they are expressed in the form of entity-relationship graphs.

Graph data are very powerful because of their ability to model arbitrary relationships between objects. The flexibility in graph representation comes at the price of greater computational complexity:

1. Graphs lack the "flat" structure of multidimensional or even contextual (e.g., time series) data. The latter is much easier to analyze with conventional models.

2. The repetition of labels among nodes leads to problems of *isomorphism* in computing similarity between graphs. This problem is NP-hard. This leads to computational challenges in similarity computation and graph matching.

The second issue is of considerable importance, because both matching and distance computation are fundamental subproblems in graph mining applications. For example, in a frequent subgraph mining application, an important subproblem is that of subgraph matching.

This chapter is organized as follows. Section 17.2 addresses the problem of matching and distance computation in graphs. Graph transformation methods for distance computation are discussed in Sect. 17.3. An important part of this section is the preprocessing methodologies, such as topological descriptors and kernel methods, that are often used for distance computation. Section 17.4 addresses the problem of pattern mining in graphs. The problem of clustering graphs is addressed in Sect. 17.5. Graph classification is addressed in Sect. 17.6. A summary is provided in Sect. 17.7.

## 17.2 Matching and Distance Computation in Graphs

The problems of matching and distance computation are closely related in the graph domain. Two graphs are said to match when a one-to-one correspondence can be established between the nodes of the two graphs, such that their labels match, and the edge presence between corresponding nodes match. The distance between such a pair of graphs is zero. Therefore, the problem of distance computation between a pair of graphs is at least as hard as that of graph matching. Matching graphs are also said to be *isomorphic*.

It should be pointed out that the term "matching" is used in two distinct contexts for graph mining, which can sometimes be confusing. For example, pairing up nodes in a single graph with the use of edges is also referred to as matching. Throughout this chapter, unless otherwise specified, our focus is not on the node matching problem, but the pairwise *graph* matching problem. This problem is also referred to as that of graph *isomorphism*.

**Definition 17.2.1 (Graph Matching and Isomorphism)** *Two graphs $G_1 = (N_1, A_1)$ and $G_2 = (N_2, A_2)$ are isomorphic if and only if a one-to-one correspondence can be found between the nodes of $N_1$ and $N_2$ satisfying the following properties:*

1. *For each pair of corresponding nodes $i \in N_1$ and $j \in N_2$, their labels are the same.*

$$l(i) = l(j)$$

2. *Let $[i_1, i_2]$ be a node-pair in $G_1$ and $[j_1, j_2]$ be the corresponding node-pair in $G_2$. Then the edge $(i_1, i_2)$ exists in $G_1$ if and only if the edge $(j_2, j_2)$ exists in $G_2$.*

The computational challenges in graph matching arise because of the repetition in node labels. For example, consider two methane molecules, illustrated in Fig. 17.2. While the unique carbon atom in the two molecules can be matched in exactly one way, the hydrogen atoms can be matched up in $4! = 24$ different ways. Two possible matchings are illustrated in Figs. 17.2a and b, respectively. In general, greater the level of label repetition in each graph is, larger the number of possible matchings will be. The number of possible matchings between a pair of graphs increases exponentially with the size of the matched graphs. For a pair of graphs containing $n$ nodes each, the number of possible matchings can be as large as $n!$. This makes the problem of matching a pair of graphs computationally very expensive.

**Lemma 17.2.1** *The problem of determining whether a matching exists between a pair of graphs, is NP-hard.*

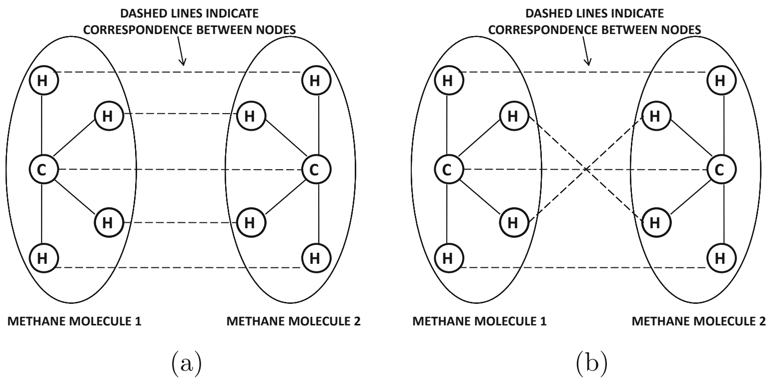


Figure 17.2: Two possible matchings between a pair of graphs representing methane molecules

The bibliographic notes contain pointers to the proof of NP-hardness. When the graphs are very large, exact matches often do not exist. However, approximate matches may exist. The level of approximation is quantified with the use of a distance function. Therefore, distance function computation between graphs is a more general problem than that of graph matching, and it is at least as difficult. This issue will be discussed in detail in the next section.

Another related problem is that of subgraph matching. Unlike the problem of exact graph matching, the query graph needs to be explicitly distinguished from the data graph in this case.

**Definition 17.2.2 (Node-Induced Subgraph)** *A node-induced subgraph of a graph $G = (N, A)$ is a graph $G_s = (N_s, A_s)$ satisfying the following properties:*

1. $N_s \subseteq N$

2. $A_s = A \cap (N_s \times N_s)$

*In other words, all the edges in the original graph $G$ between nodes in the subset $N_s \subseteq N$ are included in the subgraph $G_s$.*

A subgraph isomorphism can be defined in terms of the node-induced subgraphs. A query graph $G_q$ is a subgraph isomorphism of a data graph $G$, when it is an exact isomorphism of a node-induced subgraph of $G$.

**Definition 17.2.3 (Subgraph Matching and Isomorphism)** *A query graph $G_q = (N_q, A_q)$ is a subgraph isomorphism of the data graph $G = (N, A)$ if and only if the following conditions are satisfied:*

1. *Each node in $N_q$ should be matched to a unique node with the same label in $N$, but each node in $N$ may not necessarily be matched. For each node $i \in N_q$, there must exist a unique matching node $j \in N$, such that their labels are the same.*

$$l(i) = l(j)$$

2. *Let $[i_1, i_2]$ be a node-pair in $G_q$, and let $[j_1, j_2]$ be the corresponding node-pair in $G$, based on the matching discussed above. Then, the edge $(i_1, i_2)$ exists in $G_q$ if and only if the edge $(j_1, j_2)$ exists in $G$.*

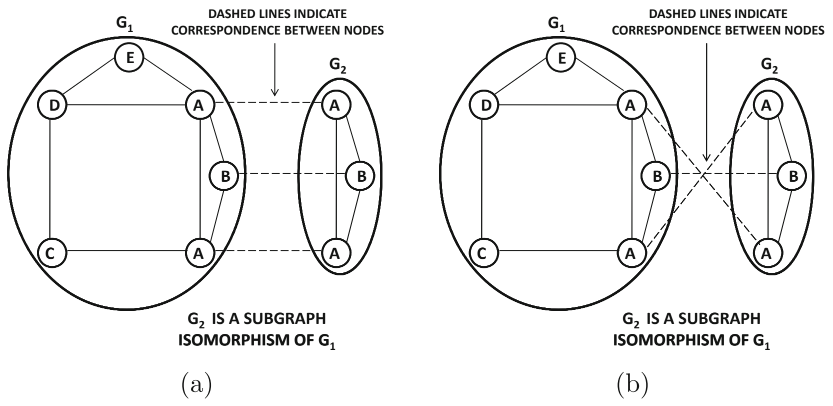

G1
DASHED LINES INDICATE CORRESPONDENCE BETWEEN NODES
G2
E
D
A
B
C
A
A
B
A
G2 IS A SUBGRAPH ISOMORPHISM OF G1
(a)
G1
DASHED LINES INDICATE CORRESPONDENCE BETWEEN NODES
G2
E
D
A
B
C
A
A
B
A
G2 IS A SUBGRAPH ISOMORPHISM OF G1
(b)

Figure 17.3: Two possible subgraph isomorphisms between a pair of graphs

The definition of subgraph isomorphism in this section assumes that *all* edges of the node-induced subgraph of the data graph are present in the query graph. In some applications, such as frequent subgraph mining, a more general definition is used, in which any subset of edges of the node-induced subgraph is also considered a subgraph isomorphism. The more general case can be handled with minor changes to the algorithm in this section. Note that the aforementioned definition allows the subgraph $G_q$ (or $G$) to be disconnected. However, for practical applications, one is usually interested only in *connected* subgraph isomorphisms. Examples of two possible subgraph matchings between a pair of nodes are illustrated in Fig. 17.3. The figure also illustrates that there are two different ways for one graph to be a subgraph of the other. The problem of exact matching is a special case of subgraph matching. Therefore, the problem of subgraph matching is NP-hard as well.

**Lemma 17.2.2** *The problem of subgraph matching is NP-hard.*

Subgraph matching is often used as a subroutine in applications such as frequent pattern mining. While the subgraph matching problem is a generalization of exact matching, the problem can be generalized even further to that of finding the maximum common subgraph (MCG) between a pair of graphs. This is because the MCG between two graphs is at most equal to the smaller of the two graphs when it is a subgraph of the larger one. The *MCG* or *maximum common isomorphism* between a pair of graphs is defined as follows.

**Definition 17.2.4 (Maximum Common Subgraph)** *A MCG between a pair of graphs $G_1 = (N_1, A_1)$ and $G_2 = (N_2, A_2)$ is a graph $G_0 = (N_0, A_0)$ that is a subgraph isomorphism of both $G_1$ and $G_2$, and for which the size of the node set $N_0$ is as large as possible.*

Because the MCG problem is a generalization of the graph isomorphism problem, it is NP-hard as well. In this section, algorithms for discovering subgraph isomorphisms and maximum common subgraphs will be presented. Subsequently, the relationship of these algorithms to that of distance computation between graphs will be discussed. Subgraph isomorphism algorithms can be designed to determine either *all* the subgraph isomorphisms between a query graph and a data graph, or a fast algorithm can be designed to determine *whether or not at least one* isomorphism exists.

### 17.2.1 Ullman's Algorithm for Subgraph Isomorphism

Ullman's algorithm is designed to determine *all* possible subgraph isomorphisms between a query graph and a data graph. It can also be used for the decision problem of determining whether or not a query graph is a subgraph isomorphism of a data graph by using an early termination criterion. Interestingly, the majority of the later graph matching algorithms are refinements of Ullman's algorithm. Therefore, this section will first present a very simplified version of Ullman's algorithm without any refinements. Subsequently, the different variations and refinements of this basic algorithm will be discussed in a separate subsection. Although the definition of subgraph isomorphisms allows the query (or data) graph to be disconnected, it is often practical and computationally expedient to focus on cases where the query and data graph are connected. Typically, small changes to the algorithm can accommodate both cases (see Exercise 14).

It will be assumed that the query graph is denoted by $G_q = (N_q, A_q)$, and the data graph is denoted by $G = (N, A)$. The first step in Ullman's algorithm is to match all possible pairs of nodes across the two graphs so that each node in the pair has the same label as the other. For each such matching pair, the algorithm expands it a node at a time with the use of a recursive search procedure. Each recursive call expands the matching subgraphs in $G_q$ and $G$ by one node. Therefore, one of the parameters of the recursive call is the current matching set $\mathcal{M}$ of node-pairs. Each element of $\mathcal{M}$ is a pair of matching nodes between $G_q$ and $G$. Therefore, when a subgraph of $m$ nodes has been matched between the two graphs, the set $\mathcal{M}$ contains $m$ matched node-pairs as follows:

$$\mathcal{M} = \{(i_1^q, i_1), (i_2^q, i_2), \ldots (i_m^q, i_m)\}$$

Here, it is assumed that the node $i_r^q$ belongs to the query graph $G_q$ and that the node $i_r$ belongs to the data graph $G$. The value of the matching set parameter $\mathcal{M}$ is initialized to the empty set at the top-level recursive call. The number of matched nodes in $\mathcal{M}$ is exactly equal to the depth of the recursive call. The recursion backtracks when either the subgraphs cannot be further matched or when $G_q$ has been fully matched. In the latter case, the matching set $\mathcal{M}$ is reported, and the recursion backtracks to the next higher level to discover other matchings. In cases where it is not essential to determine *all possible* matchings between the pair of graphs, it is also possible to terminate the algorithm at this point. This particular exposition, however, assumes that all possible matchings need to be determined.

A simplified version of Ullman's algorithm is illustrated in Fig. 17.4. The algorithm is structured as a recursive approach that explores the space of all possible matchings between the two graphs. The input to the algorithm is the query graph $G_q$ and the data graph $G$. An additional parameter $\mathcal{M}$ of this recursive call is a set containing the current matching node-pairs. While the set $\mathcal{M}$ is empty at the top-level call made by the analyst, this is not the case at lower levels of the recursion. The cardinality of $\mathcal{M}$ is exactly equal to the depth of the recursion. This is because one matching node-pair is added to $\mathcal{M}$ in each recursive call. Strictly speaking, the recursive call returns all the subgraph isomorphisms under the constraint that the matching corresponding to $\mathcal{M}$ must be respected.

The first step of the recursive procedure is to check whether the size of $\mathcal{M}$ is equal to the number of nodes in the query graph $G_q$. If this is indeed the case, then the algorithm reports $\mathcal{M}$ as a successful subgraph matching and backtracks out of the recursion to the next higher level to explore other matchings. Otherwise, the algorithm tries to determine further matching node-pairs to add to $\mathcal{M}$. This is the *candidate generation step*. In this

```
Algorithm SubgraphMatch(Query Graph: $G_q$, Data Graph: $G$,
        Current Partially Matched Node Pairs: $\mathcal{M}$)
begin
  if ($|\mathcal{M}| = |N_q|$) then return successful match $\mathcal{M}$;
  (Case when $|\mathcal{M}| < |N_q|$)
  $\mathcal{C}$ = Set of all label matching node pairs from $(G_q, G)$ not in $\mathcal{M}$;
  Prune $\mathcal{C}$ using heuristic methods; (Optional efficiency optimization)
  for each pair $(i_q, i) \in \mathcal{C}$ do
    if $\mathcal{M} \cup \{(i_q, i)\}$ is a valid partial matching
        then SubgraphMatch($G_q, G, \mathcal{M} \cup \{(i_q, i)\}$);
  endfor
end
```

Figure 17.4: The basic template of Ullman's algorithm

step, all possible label matching node-pairs between $G_q$ and $G$, which are not already in $\mathcal{M}$, are used to construct the candidate match set $\mathcal{C}$.

Because the number of candidate match extensions can be large, it is often desirable to prune them heuristically by using specific properties of the data graph and query graph. Some examples of such heuristics will be presented later. After the pruned set $\mathcal{C}$ has been generated, node-pairs $(i_q, i) \in \mathcal{C}$ are selected one by one, and it is checked whether they can be added to $\mathcal{M}$ to create a valid (partial) matching between the two graphs. For $\mathcal{M} \cup \{(i_q, i)\}$ to be a valid partial matching, if $i_q \in N_q$ is incident on any already matched node $j_q$ in $G_q$, then $i$ must also be incident on the matched counterpart $j$ of $j_q$ in $G$, and vice versa. If a valid partial matching exists, then the procedure is called recursively with the partial matching $\mathcal{M} \cup \{(i_q, i)\}$. After iterating through all such candidate extensions with corresponding recursive calls, the algorithm backtracks to the next higher level of the recursion.

It is not difficult to see that the procedure has exponential complexity in terms of its input size, and it is especially sensitive to the query graph size. This high complexity is because the depth of the recursion can be of the order of the query graph size, and the number of recursive branches at each level is equal to the number of matching node-pairs. Clearly, unless the number of candidate extensions is carefully controlled with more effective candidate generation and pruning, the approach will be extremely slow.

#### 17.2.1.1 Algorithm Variations and Refinements

Although the basic matching algorithm was originally proposed by Ullman, this template has been used extensively by different matching algorithms. The different algorithms vary from one another in terms of how the size of the candidate matched pairs is restricted with careful pruning. The use of carefully selected candidate sets has a significant impact on the efficiency of the algorithm. Most pruning methods rely on a number of natural constraints that are always satisfied by two graphs in a subgraph isomorphism relationship. Some common pruning rules are as follows:

1. *Ullman's algorithm:* This algorithm uses a simple pruning rule. All node-pairs $(i_q, i)$ are pruned from $\mathcal{C}$ in the pruning step if the degree of $i$ is less than $i_q$. This is because the degree of every matching node in the query subgraph needs to be no larger than the degree of its matching counterpart in the data graph.

2. *VF2 algorithm:* In the *VF2* algorithm, those candidates $(i_q, i)$ are pruned if $i_q$ is not connected to already matched nodes in $G_q$ (i.e., nodes of $G_q$ included in $\mathcal{M}$). Subsequently, the pruning step also removes those node-pairs $(i_q, i)$ in which $i$ is not connected to the matched nodes in the data graph $G$. These pruning rules assume that the query and data graphs are connected. The algorithm also compares the number of neighbor nodes of each of $i$ and $i_q$ that are connected to nodes in $\mathcal{M}$ but are not included in $\mathcal{M}$. The number of such nodes in the data graph must be no smaller than the number of such nodes in the query graph. Finally, the number of neighbor nodes of each of $i$ and $i_q$ that are not directly connected to nodes in $\mathcal{M}$ are compared. The number of such nodes in the data graph must be no smaller than the number of such nodes in the query graph.

3. *Sequencing optimizations:* The effectiveness of the pruning steps is sensitive to the order in which nodes are added to the matching set $\mathcal{M}$. In general, nodes with rarer labels in the query graph should be selected first in the exploration of different candidate pairs in $\mathcal{C}$. Rarer labels can be matched in fewer ways across graphs. Early exploration of rare labels leads to exploration of more relevant partial matches $\mathcal{M}$ at the earlier levels of the recursion. This also helps the pruning effectiveness. Enhanced versions of *VF2* and *QuickSI* combine node sequencing and the aforementioned node pruning steps.

The reader is referred to the bibliographic notes for details of these algorithms. The definition of subgraph isomorphism in this section assumes that *all* edges of the node-induced subgraph of the data graph are present in the query graph. In some applications, such as frequent subgraph mining, a more general definition is used, in which any subset of edges of the node-induced subgraph is also considered a subgraph isomorphism. The more general case can be solved with minor changes to the basic algorithm in which the criteria to generate candidates and validate them are both relaxed appropriately.

### 17.2.2 Maximum Common Subgraph (MCG) Problem

The MCG problem is a generalization of the subgraph isomorphism problem. The MCG between two graphs is at most equal to the smaller of the two, when one is a subgraph of the other. The basic principles of subgraph isomorphism algorithms can be extended easily to the MCG isomorphism problem. The following will discuss the extension of the Ullman algorithm to the MCG problem. The main differences between these methods are in terms of the pruning criteria and the fact that the maximum common subgraph is continuously tracked over the course of the algorithm as the search space of subgraphs is explored.

The recursive exploration process of the MCG algorithm is identical to that of the subgraph isomorphism algorithm. The algorithm is illustrated in Fig. 17.5. The two input graphs are denoted by $G_1$ and $G_2$, respectively. As in the case of subgraph matching, the current matching in the recursive exploration is denoted by the set $\mathcal{M}$. For each matching node-pair $(i_1, i_2) \in \mathcal{M}$, it is assumed that $i_1$ is drawn from $G_1$, and $i_2$ is drawn from $G_2$. Another input parameter to the algorithm is the current best (largest) matching set of node-pairs $\mathcal{M}_{best}$. Both $\mathcal{M}$ and $\mathcal{M}_{best}$ are initialized to *null* in the initial call made to the recursive algorithm by the analyst. Strictly speaking, each recursive call determines the best matching under the constraint that the pairs in $\mathcal{M}$ must be matched. This is the reason that this parameter is set to *null* at the top-level recursive call. However, in lower level calls, the value of $\mathcal{M}$ is not *null*.

```
Algorithm MCG(Graphs: $G_1, G_2$, Current Partially Matched Pairs: $\mathcal{M}$,
              Current Best Match: $\mathcal{M}_{best}$)
begin
  $\mathcal{C}$ = Set of all label matching node pairs from $(G_1, G_2)$ not in $\mathcal{M}$;
  Prune $\mathcal{C}$ using heuristic methods; (Optional efficiency optimization)
  for each pair $(i_1, i_2) \in \mathcal{C}$ do
    if $\mathcal{M} \cup \{(i_1, i_2)\}$ is a valid matching
       then $\mathcal{M}_{best}$ =MCG($G_1, G_2, \mathcal{M} \cup \{(i_1, i_2)\}, \mathcal{M}_{best}$);
  endfor
  if $(|\mathcal{M}| > |\mathcal{M}_{best}|)$ then return($\mathcal{M}$) else return($\mathcal{M}_{best}$);
end
```

Figure 17.5: Maximum common subgraph (MCG) algorithm

As in the case of the subgraph isomorphism algorithm, the candidate matching node-pairs are explored recursively. The same steps of candidate extension and pruning are used in the MCG algorithm, as in the case of the subgraph isomorphism problem. However, some of the pruning steps used in the subgraph isomorphism algorithm, which are based on *subgraph* assumptions, can no longer be used. For example, in the MCG algorithm, a matching node-pair $(i_1, i_2)$ in $\mathcal{M}$ no longer needs to satisfy the constraint that the degree of a node in one graph is greater or less than that of its matching node in the other. Because of the more limited pruning in the maximum common subgraph problem, it will explore a larger search space. This is intuitively reasonable, because the maximum common subgraph problem is a more general one than subgraph isomorphism. However, some optimizations such as expanding only to connected nodes, and sequencing optimizations such as processing rare labels earlier, can still be used.

The largest common subgraph found so far is tracked in $\mathcal{M}_{best}$. At the end of the procedure, the largest matching subgraph found so far is returned by the algorithm. It is also relatively easy to modify this algorithm to determine all possible MCGs. The main difference is that all the current MCGs can be dynamically tracked instead of tracking a single MCG.

### 17.2.3 Graph Matching Methods for Distance Computation

Graph matching methods are closely related to distance computation between graphs. This is because pairs of graphs that share large subgraphs in common are likely to be more similar. A second way to compute distances between graphs is by using the *edit distance*. The edit distance in graphs is analogous to the notion of the edit distance in strings. Both these methods will be discussed in this section.

#### 17.2.3.1 MCG-based Distances

When two graphs share a large subgraph in common, it is indicative of similarity. There are several ways of transforming the MCG size into a distance value. Some of these distance definitions have also been demonstrated to be *metrics* because they are nonnegative, symmetric, and satisfy the triangle inequality. Let the MCG of graphs $G_1$ and $G_2$ be denoted by $MCS(G_1, G_2)$ with a size of $|MCS(G_1, G_2)|$. Let the sizes of the graphs $G_1$ and $G_2$

be denoted by $|G_1|$ and $|G_2|$, respectively. The various distance measures are defined as a function of these quantities.

1. *Unnormalized non-matching measure:* The unnormalized non-matching distance measure $U(G_1, G_2)$ between two graphs is defined as follows:

$$U(G_1, G_2) = |G_1| + |G_2| - 2 \cdot |MCS(G_1, G_2)| \tag{17.1}$$

This is equal to the number of non-matching nodes between the two graphs because it subtracts out the number of matching nodes $|MCS(G_1, G_2)|$ from each of $|G_1|$ and $|G_2|$ and then adds them up. This measure is unnormalized because the value of the distance depends on the raw size of the underlying graphs. This is not desirable because it is more difficult to compare distances between pairs of graphs of varying size. This measure is more effective when the different graphs in the collection are of approximately similar size.

2. *Union-normalized distance:* The distance measure lies in the range of $(0, 1)$, and is also shown to be a metric. The union-normalized measure $UDist(G_1, G_2)$ is defined as follows:

$$UDist(G_1, G_2) = 1 - \frac{|MCS(G_1, G_2)|}{|G_1| + |G_2| - |MCS(G_1, G_2)|} \tag{17.2}$$

This measure is referred to as the union-normalized distance because the denominator contains the number of nodes in the union of the two graphs. A different way of understanding this measure is that it normalizes the number of non-matching nodes $U(G_1, G_2)$ between the two graphs (unnormalized measure) with the number of nodes in the union of the two graphs.

$$UDist(G_1, G_2) = \frac{\text{Non-matching nodes between } G_1 \text{ and } G_2}{\text{Union size of } G_1 \text{ and } G_2}$$

One advantage of this measure is that it is intuitively easier to interpret. Two perfectly matching graphs will have a distance of 0 from one another, and two perfectly non-matching graphs will have a distance of 1.

3. *Max-normalized distance:* This distance measure also lies in the range $(0, 1)$. The *max-normalized distance* $MDist(G_1, G_2)$ between two graphs $G_1$ and $G_2$ is defined as follows:

$$MDist(G_1, G_2) = 1 - \frac{|MCS(G_1, G_2)|}{\max\{|G_1|, |G_2|\}} \tag{17.3}$$

The main difference from the union-normalized distance is that the denominator is normalized by the maximum size of the two graphs. This distance measure is a metric because it satisfies the triangle inequality. The measure is also relatively easy to interpret. Two perfectly matching graphs will have a distance of 0 from one another, and two perfectly non-matching graphs will have a distance of 1.

These distance measures can be computed effectively only for small graphs. For larger graphs, it becomes computationally too expensive to evaluate these measures because of the need to determine the maximum common subgraph between the two graphs.

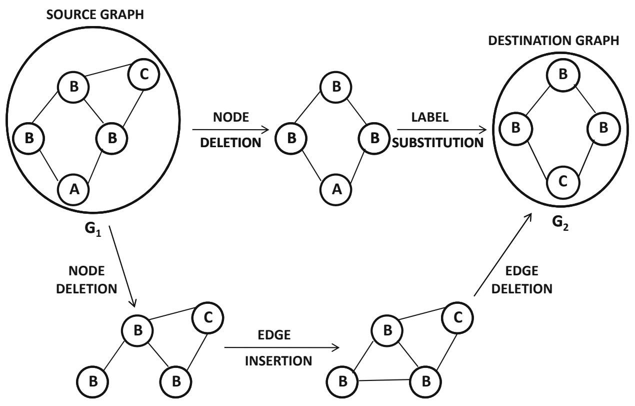


Figure 17.6: Example of two possible edits paths between graphs $G_1$ and $G_2$

#### 17.2.3.2 Graph Edit Distance

The graph edit distance is analogous to the string edit distance, discussed in Chap. 3. The main difference is that the edit operations are specific to the graph domain. The edit distances can be applied to the nodes, the edges, or the labels. In the context of graphs, the admissible operations include (a) the insertion of nodes, (b) the deletion of nodes, (c) the label-substitution of nodes, (d) the insertion of edges, and (e) the deletion of edges. Note that the deletion of a node includes automatic deletion of all its incident edges. Each edit operation has an edit cost associated with it that is defined in an application-specific manner. In fact, the problem of learning edit costs is a challenging issue in its own right. For example, one way of learning edit costs is to use supervised distance function learning methods discussed in Chap. 3. The bibliographic notes contain pointers to some of these algorithms.

An example of two possible edit paths between graphs $G_1$ and $G_2$ is illustrated in Fig. 17.6. Note that the two paths will have different costs, depending on the costs of the constituent operations. For example, if the cost of label-substitution is very high compared to that of edge insertions and deletions, it may be more effective to use the second (lower) path in Fig. 17.6. For large and complex pairs of graphs, an exponential number of possible edit paths may exist. The edit distance $Edit(G_1, G_2)$ between two graphs is equal to the minimum cost of transforming graph $G_1$ to $G_2$ with a series of edit operations.

**Definition 17.2.5 (Graph Edit Distance)** *The graph edit distance $Edit(G_1, G_2)$ is the minimum cost of the edit operations to be applied to graph $G_1$ in order to transform it to graph $G_2$.*

Depending on the costs of the different operations, the edit distance is not necessarily symmetric. In other words, $Edit(G_1, G_2)$ can be different from $Edit(G_2, G_1)$. Interestingly, the edit distance is closely related to the problem of determining MCGs. In fact, for some special choices of costs, the edit distance can be shown to be equivalent to distance measures based on the maximum common subgraph. This implies that the edit-distance computation for graphs is NP-hard as well. The edit distance can be viewed as the cost of an *error-tolerant graph isomorphism*, where the "errors" are quantified in terms of the cost of edit operations. As discussed in Chap. 3, the edit-distance computation for strings and sequences can be solved polynomially using dynamic programming. The case of graphs is more difficult because it belongs to the class of NP-hard problems.

The close relationship between edit-distance computation and the MCG problem is reflected in the similar structure of their corresponding algorithms. As in the case of the maximum common subgraph problem, a recursive tree-search procedure can be used to compute the edit distance. In the following, the basic procedure for computing the edit distance will be described. The bibliographic notes contain pointers to various enhancements of this procedure.

An interesting property of the edit-distance is that it can be computed by exploring only those edit sequences in which any and all node insertion operations (together with their incident edge insertions) are performed at the end of the edit sequence. Therefore, the edit-distance algorithm maintains a series of edits $\mathcal{E}$ that are the operations to be applied to graph $G_1$ to transform it into a *subgraph isomorphism* $G'_1$ of the graph $G_2$. By trivially adding the unmatched nodes of $G_2$ to $G'_1$ and corresponding incident edges as the final step, it is possible to create $G_2$. Therefore, the initial part of sequence $\mathcal{E}$, without the last step, does not contain any node insertions at all. In other words, the initial part of sequence $\mathcal{E}$ may contain node deletions, node label-substitutions, edge additions, and edge deletions. An example of such an edit sequence is as follows:

$$\mathcal{E} = \text{Delete}(i_1),\ \text{Insert}(i_2, i_5),\ \text{Label-Substitute}(i_4, A \Rightarrow C),\ \text{Delete}(i_2, i_6)$$

This edit sequence illustrates the deletion of a node, followed by addition of the new edge $(i_2, i_5)$. The label of node $i_4$ is substituted from $A$ to $C$. Then, the edge $(i_2, i_6)$ is deleted. The total cost of an edit sequence $\mathcal{E}$ from $G_1$ to a subgraph isomorphism $G'_1$ of $G_2$ is equal to the sum of the edit costs of all the operations in $\mathcal{E}$, together with the cost of the node insertions and incident edge insertions that need to be performed on $G'_1$ to create the final graph $G_2$.

The correctness of such an approach relies on the fact that it is always possible to arrange the optimal edit path sequence, so that the insertion of the nodes and their incident edges is performed *after* all other edge operations, node deletions and label-substitutions that transform $G_1$ to a subgraph isomorphism of $G_2$. The proof of this property follows from the fact that any *optimal* edit sequence can be reordered to push the insertion of nodes (and their incident edges) to the end, as long as an inserted node is not associated with any other edit operations (node or incident edge deletions, or label-substitutions). It is also easy to show that any edit path in which newly added nodes or edges are deleted will be suboptimal. Furthermore, an inserted node never needs to be label-substituted in an optimal path because the correct label can be set at the time of node insertion.

The overall recursive procedure is illustrated in Fig. 17.7. The inputs to the algorithm are the source and target graphs $G_1$ and $G_2$, respectively. In addition, the current edit sequence $\mathcal{E}$ being examined for further extension, and the best (lowest cost) edit sequence $\mathcal{E}_{best}$ found so far, are among the input parameters to the algorithm. These input parameters are useful for passing data between recursive calls. The value of $\mathcal{E}$ is initialized to *null* in the top-level call. While the value of $\mathcal{E}$ is *null* at the beginning of the algorithm, new edits are appended to it in each recursive call. Further recursive calls are executed with this extended sequence as the input parameter. The value of the parameter $\mathcal{E}_{best}$ at the top-level call is set to a trivial sequence of edit operations in which all nodes of $G_1$ are deleted and then all nodes and edges of $G_2$ are added.

The recursive algorithm first discovers the sequence of edits $\mathcal{E}$ that transforms the graph $G_1$ to a subgraph isomorphism $G'_1$ of $G_2$. After this phase, the trivial sequence of node/edge insertion edits that convert $G'_1$ to $G_2$ is padded at the end of $\mathcal{E}$. This step is shown in Fig. 17.7 just before the return condition in the recursive call. Because of this final padding step, the

```
Algorithm EditDistance(Graphs: $G_1, G_2$, Current Partial Edit Sequence: $\mathcal{E}$,
    Best Known Edit Sequence: $\mathcal{E}_{best}$)
begin
  if ($G_1$ is subgraph isomorphism of $G_2$) then begin
    Add insertion edits to $\mathcal{E}$ that convert $G_1$ to $G_2$;
    return($\mathcal{E}$);
  end;
  $\mathcal{C}$ = Set of all possible edits to $G_1$ excluding node-insertions;
  Prune $\mathcal{C}$ using heuristic methods; (Optional efficiency optimization)
  for each edit operation $e \in \mathcal{C}$ do
  begin
    Apply $e$ to $G_1$ to create $G_1'$;
    Append $e$ to $\mathcal{E}$ to create $\mathcal{E}'$;
    $\mathcal{E}_{current}$ = EditDistance($G_1', G_2, \mathcal{E}', \mathcal{E}_{best}$);
    if ($Cost(\mathcal{E}_{current}) < Cost(\mathcal{E}_{best})$) then $\mathcal{E}_{best} = \mathcal{E}_{current}$;
  endfor
  return($\mathcal{E}_{best}$);
end
```

Figure 17.7: Graph edit distance algorithm

cost of these trivial edits is always included in the cost of the edit sequence $\mathcal{E}$, which is denoted by $Cost(\mathcal{E})$.

The overall structure of the algorithm is similar to that of the MCG algorithm of Fig. 17.5. In each recursive call, it is first determined if $G_1$ is a subgraph isomorphism of $G_2$. If so, the algorithm immediately returns the current set of edits $\mathcal{E}$ after the incorporation of trivial node or edge insertions that can transform $G_1$ to $G_2$. If $G_1$ is not a subgraph isomorphism of $G_2$, then the algorithm proceeds to extend the partial edit path $\mathcal{E}$. A set of candidate edits $\mathcal{C}$ is determined, which when applied to $G_1$ *might* reduce the distance to $G_2$. In practice, these candidate edits $\mathcal{C}$ are determined heuristically because the problem of knowing the precise impact of an edit on the distance is almost as difficult as that of computing the edit distance. The simplest way of choosing the candidate edits is to consider all possible unit edits excluding node insertions. These candidate edits might be node deletions, label-substitutions and edge operations (both insertions and deletions). For a graph with $n$ nodes, the total number of node-based candidate operations is $O(n)$, and the number of edge-based candidate operations is $O(n^2)$. It is possible to heuristically prune many of these candidate edits if it can be immediately determined that such edits can never be part of an optimal edit path. In fact, some of the pruning steps are essential to ensure finite termination of the algorithm. Some key pruning steps are as follows:

1. An edge insertion cannot be appended to the current partial edit sequence $\mathcal{E}$, if an edge deletion operation between the same pair of nodes already exists in the current partial edit path $\mathcal{E}$. Similarly, an edge which was inserted earlier cannot be deleted. An optimal edit path can never include such pairs of edits with zero net effect. This pruning step is necessary to ensure finite termination.

2. The label of a node cannot be substituted, if the label-substitution of that node exists in the current partial edit path $\mathcal{E}$. Repetitive label-substitutions of the same node are obviously suboptimal.

3. An edge can be inserted between a pair of nodes in $G_1$, only if at least one edge exists in $G_2$ between two nodes with the same labels.

4. A candidate edit should not be considered, if adding it to $\mathcal{E}$ would immediately increase the cost of $\mathcal{E}$ beyond that of $\mathcal{E}_{best}$.

5. Many other sequencing optimizations are possible for prioritizing between candidate edits. For example, all node deletions can be performed before all label-substitutions. It can be shown that the optimal edit sequence can always be arranged in this way. Similarly, label-substitutions which change the overall distribution of labels closer to the target graph may be considered first. In general, it is possible to associate a "goodness-function" with an edit, which heuristically quantifies its likelihood of finding a good edit path, when included in $\mathcal{E}$. Finding good edit paths early will ensure better pruning performance according to the aforementioned criterion (4).

The main difference among various recursive search algorithms is to use different heuristics for candidate ordering and pruning. Readers are referred to the bibliographic notes at the end of this chapter for pointers to some of these methods. After the pruned candidate edits have been determined, each of these is applied to $G_1$ to create $G'_1$. The procedure is recursively called with the pair $(G'_1, G_2)$, and an augmented edit sequence $\mathcal{E}'$. This procedure returns the best edit sequence $\mathcal{E}_{current}$ which has a prefix of $\mathcal{E}'$. If the cost of $\mathcal{E}_{current}$ is better than $\mathcal{E}_{best}$ (including trivial post-processing insertion edits for full matching), then $\mathcal{E}_{best}$ is updated to $\mathcal{E}_{current}$. At the end of the procedure, $\mathcal{E}_{best}$ is returned.

The procedure is guaranteed to terminate because repetitions in node label-substitutions and edge deletions are avoided in $\mathcal{E}$ by the pruning steps. Furthermore, the number of nodes in the edited graph is monotonically non-increasing as more edits are appended to $\mathcal{E}$. This is because $\mathcal{E}$ does not contain node insertions except at the end of the recursion. For a graph with $n$ nodes, there are at most $\binom{n}{2}$ non-repeating edge additions and deletions and $O(n)$ node deletions and label-substitutions that can be performed. Therefore, the recursion has a finite depth of $O(n^2)$ that is also equal to the maximum length of $\mathcal{E}$. This approach t has exponential complexity in the worst case. Edit distances are generally expensive to compute, unless the underlying graphs are small.

## 17.3 Transformation-Based Distance Computation

The main problem with the distance measures of the previous section is that they are computationally impractical for larger graphs. A number of heuristic and kernel-based methods are used to transform the graphs into a space in which distance computations are more efficient. Interestingly, some of these methods are also *qualitatively* more effective because of their ability to focus on the relevant portions of the graphs.

### 17.3.1 Frequent Substructure-Based Transformation and Distance Computation

The intuition underlying this approach is that frequent graph patterns encode key properties of the graph. This is true of many applications. For example, the presence of a benzene ring (see Fig. 17.1) in a chemical compound will typically result in specific properties. Therefore, the properties of a graph can often be described by the presence of specific families of structures in it. This intuition suggests that a meaningful way of semantically describing the graph is in terms of its family of frequent substructures. Therefore, a transformation approach is used in which a text-like vector-space representation is created from each graph. The steps are as follows:

1. Apply frequent subgraph mining methods discussed in Sect. 17.4 to discover frequent subgraph patterns in the underlying graphs. This results in a "lexicon" in terms of which the graphs are represented. Unfortunately, the size of this lexicon is rather large, and many subgraphs may be redundant because of similarity to one another.

2. Select a subset of subgraphs from the subgraphs found in the first step to reduce the overlap among the frequent subgraph patterns. Different algorithms may vary in this step by using only frequent maximal subgraphs, or selecting a subset of graphs that are sufficiently nonoverlapping with one another. Create a new feature $f_i$ for each frequent subgraph $S_i$ that is finally selected. Let $d$ be the total number of frequent subgraphs (features). This is the lexicon size in terms of which a text-like representation will be constructed.

3. For each graph $G_i$, create a vector-space representation in terms of the features $f_1 \ldots f_d$. Each graph contains the features, corresponding to the subgraphs that it contains. The frequency of each feature is the number of occurrences of the corresponding subgraph in the graph $G_i$. It is also possible to use a binary representation by only considering presence or absence of subgraphs, rather than frequency of presence. The tf-idf normalization may be used on the vector-space representation, as discussed in Chap. 13.

After the transformation has been performed, any of the text similarity functions can be used to compute distances between graph objects. One advantage of using this approach is that it can be paired up with a conventional text index, such as the inverted index, for efficient retrieval. The bibliographic notes contain pointers to some of these methods.

This broader approach can also be used for feature transformation. Therefore, any data mining algorithm from the text domain can be applied to graphs using this approach. Later, it will be discussed how this transformation approach can be used in a more direct way by graph mining algorithms such as clustering. The main disadvantage of this approach is that subgraph isomorphism is an intermediate step in frequent substructure discovery. Therefore, the approach has exponential complexity in the worst case. Nevertheless, many fast approximations are often used to provide more efficient results without a significant loss in accuracy.

## 17.3.2 Topological Descriptors

Topological descriptors convert structural graphs to multidimensional data by using quantitative measures of important structural characteristics as dimensions. After the conversion has been performed, multidimensional data mining algorithms can be used on the transformed representation. This approach enables the use of a wide variety of multidimensional data mining algorithms in graph-based applications. The drawback of the approach is that the structural information is lost. Nevertheless, topological descriptors have been shown to retain important properties of graphs in the chemical domain, and are therefore used quite frequently. In general, the utility of topological descriptors in graph mining is highly domain specific. It should be pointed out that topological descriptors share a number of conceptual similarities with the frequent subgraph approach in the previous section. The

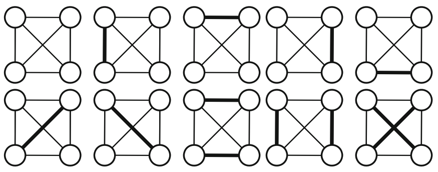


Figure 17.8: The Hosoya index for a clique of four nodes

main difference is that carefully chosen topological parameters are used to define the new feature space instead of frequent subgraphs.

Most topological descriptors are graph specific, whereas a few are node-specific. The vector of node-specific descriptors can sometimes describe the graph quite well. Node specific descriptors can also be used for enriching the labels of the nodes. Some common examples of topological descriptors are as follows:

1. *Morgan index:* This is a node-specific index that is equal to the $k$th order degree of a node. In other words, the descriptor is equal to the number of nodes reachable from the node within a distance of $k$. This is one of the few descriptors that describes nodes, rather than the complete graph. The node-specific descriptors can also be converted to a graph-specific descriptor by using the frequency histogram of the Morgan index over different nodes.

2. *Wiener index:* The Wiener index is equal to the sum of the pairwise shortest path distances between all pairs of nodes. It is therefore required to compute the all-pairs shortest path distance between different pairs of nodes.

$$W(G) = \sum_{i,j \in G} d(i,j) \tag{17.4}$$

   The Wiener index has known relationships with the chemical properties of compounds. The motivating reason for this index was the fact that it was known to be closely correlated with the boiling points of alkane molecules [511]. Later, the relationship was also shown for other properties of some families of molecules, such as their density, surface tension, viscosity, and van der Waal surface area. Subsequently, the index has also been used for applications beyond the chemical domain.

3. *Hosoya index:* The Hosoya index is equal to the number of valid pairwise node matchings in the graph. Note that the word "matching" refers to node–node matching within the same graph, rather than graph–graph matching. The matchings do not need to be maximal matchings, and even the empty matching is counted as one of the possibilities. The determination of the Hosoya index is #P-complete because an exponential number of possible matchings may exist in a graph, especially when it is dense. For example, as illustrated in Fig. 17.8, the Hosoya index for a complete graph (clique) of only four nodes is 10. The Hosoya index is also referred to as the Z-index.

4. *Estrada index:* This index is particularly useful in chemical applications for measuring the degree of protein folding. If $\lambda_1 \dots \lambda_n$ are the eigenvalues of the adjacency matrix of graph $G$, then the Estrada index $E(G)$ is defined as follows:

$$E(G) = \sum_{i=1}^{n} e^{\lambda_i} \tag{17.5}$$

5. *Circuit rank:* The circuit rank $C(G)$ is equal to the minimum number of edges that need to be removed from a graph in order to remove all cycles. For a graph with $m$ edges, $n$ nodes, and $k$ connected components, this number is equal to $(m - n + k)$. The circuit rank is also referred to as the cyclomatic number. The cyclomatic number provides insights into the connectivity level of the graph.

6. *Randic index:* The Randic index is equal to the pairwise sum of bond contributions. If $\nu_i$ is the degree of vertex $i$, then the Randic index $R(G)$ is defined as follows:

$$R(G) = \sum_{i,j \in G} 1/\sqrt{\nu_i \cdot \nu_j} \tag{17.6}$$

   The Randic index is also known as the molecular connectivity index. This index is often used in the context of larger organic chemical compounds in order to evaluate their connectivity. The Randic index can be combined with the circuit rank $C(G)$ to yield the Balaban index $B(G)$:

$$B(G) = \frac{m \cdot R(G)}{C(G) + 1} \tag{17.7}$$

   Here, $m$ is the number of edges in the network.

Most of these indices have been used quite frequently in the chemical domain because of their ability to capture different properties of chemical compounds.

### 17.3.3 Kernel-Based Transformations and Computation

Kernel-based methods can be used for faster similarity computation than is possible with methods such as MCG-based or edit-based measures. Furthermore, these similarity computation methods can be used directly with support vector machine (SVM) classifiers. This is one of the reasons that kernel methods are very popular in graph classification.

Several kernels are used frequently in the context of graph mining. The following contains a discussion of the more common ones. The kernel similarity $K(G_i, G_j)$ between a pair of graphs $G_i$ and $G_j$ is the dot product of the two graphs after hypothetically transforming them to a new space, defined by the function $\Phi(\cdot)$.

$$K(G_i, G_j) = \Phi(G_i) \cdot \Phi(G_j) \tag{17.8}$$

In practice, the value of $\Phi(\cdot)$ is not defined directly. Rather, it is defined indirectly in terms of the kernel similarity function $K(\cdot, \cdot)$. There are various ways of defining the kernel similarity.

#### 17.3.3.1 Random Walk Kernels

In random walk kernels, the idea is to compare the label sequences induced by random walks in the two graphs. Intuitively, two graphs are similar if many sequences of labels created by random walks between pairs of nodes are similar as well. The main computational challenge is that there are an exponential number of possible random walks between pairs of nodes. Therefore, the first step is to defined a primitive kernel function $k(s_1, s_2)$ between a pair of node sequences $s_1$ (from $G_1$) and $s_2$ (from $G_2$). The simplest kernel is the identity kernel:

$$k(s_1, s_2) = I(s_1 = s_2) \tag{17.9}$$

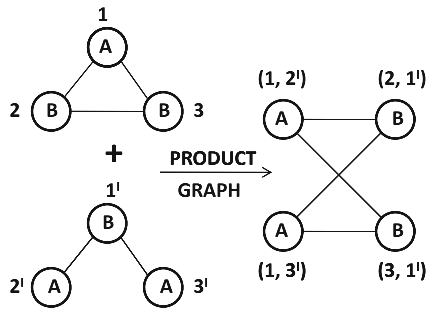


Figure 17.9: Example of the product graph

Here, $I(\cdot)$ is the indicator function that takes the value of 1 when the two sequences are the same and 0 otherwise. Then, the overall kernel similarity $K(G_1, G_2)$ is defined as the sum of the probabilities of all the primitive sequence kernels over all possible walks:

$$K(G_1, G_2) = \sum_{s_1, s_2} p(s_1|G_1) \cdot p(s_2|G_2) \cdot k(s_1, s_2) \tag{17.10}$$

Here, $p(s_i|G_i)$ is the probability of the random walk sequence $s_i$ in the graph $G_i$. Note that this kernel similarity value will be higher when the same label sequences are used by the two graphs. A key challenge is to compute these probabilities because there are an exponential number of walks of a specific length, and the length of a walk may be any value in the range $(1, \infty)$.

The random walk kernel is computed using the notion of a *product graph* $G_X$ between $G_1$ and $G_2$. The product graphs are constructed by defining a vertex $[u_1, u_2]$ between each pair of label matching vertices $u_1$ and $u_2$ in the graphs $G_1$ and $G_2$, respectively. An edge is added between a pair of vertices $[u_1, u_2]$ and $[v_1, v_2]$ in the product graph $G_X$ if and only an edge exists between the corresponding nodes in *both* the individual graphs $G_1$ and $G_2$. In other words, the edge $(u_1, v_1)$ must exist in $G_1$ and the edge $(u_2, v_2)$ must exist in $G_2$. An example of a product graph is illustrated in Fig. 17.9. Note that each walk in the product graph corresponds to a pair of label-matching sequence of vertices in the two graphs $G_1$ and $G_2$. Then, if $A$ is the binary adjacency matrix of the product graph, then the entries of $A^k$ provide the number of walks of length $k$ between the different pairs of vertices. Therefore, the total weighted number of walks may be computed as follows:

$$K(G_1, G_2) = \sum_{i,j} \sum_{k=1}^{\infty} \lambda^k [A^k]_{ij} = \overline{e}^T (I - \lambda A)^{-1} \overline{e} \tag{17.11}$$

Here, $\overline{e}$ is an $|G_X|$-dimensional column vector of 1s, and $\lambda \in (0, 1)$ is a discount factor. The discount factor $\lambda$ should always be smaller than the inverse of the largest eigenvalue of $A$ to ensure convergence of the infinite summation. Another variant of the random walk kernel is as follows:

$$K(G_1, G_2) = \sum_{i,j} \sum_{k=1}^{\infty} \frac{\lambda^k}{k!} [A^k]_{ij} = \overline{e}^T \exp(\lambda A) \overline{e} \tag{17.12}$$

When the graphs in a collection are widely varying in size, the kernel functions of Eqs. 17.11 and 17.12 should be further normalized by dividing with $|G_1| \cdot |G_2|$. Alternatively, in some probabilistic versions of the random walk kernel, the vectors $\overline{e}^T$ and $\overline{e}$ are replaced with starting and stopping probabilities of the random walk over various nodes in the product graph. This computation is quite expensive, and may require as much as $O(n^6)$ time.

#### 17.3.3.2 Shortest-Path Kernels

In the shortest-path kernel, a primitive kernel $k_s(i_1, j_1, i_2, i_2)$ is defined on node-pairs $[i_1, j_1] \in G_1$ and $[i_2, j_2] \in G_2$. There are several ways of defining the kernel function $k_s(i_1, i_2, j_1, j_2)$. A simple way of defining the kernel value is to set it to 1 when the distance $d(i_1, i_2) = d(j_1, j_2)$, and 0, otherwise.

Then, the overall kernel similarity is equal to the sum of all primitive kernels over different quadruplets of nodes:

$$K(G_1, G_2) = \sum_{i_1, i_2, j_1, j_2} k_s(i_1, i_2, j_1, j_2) \tag{17.13}$$

The shortest-path kernel may be computed by applying the all-pairs shortest-path algorithm on each of the graphs. It can be shown that the complexity of the kernel computation is $O(n^4)$. Although this is still quite expensive, it may be practical for small graphs, such as chemical compounds.

## 17.4 Frequent Substructure Mining in Graphs

Frequent subgraph mining is a fundamental building block for graph mining algorithms. Many of the clustering, classification, and similarity search techniques use frequent substructure mining as an intermediate step. This is because frequent substructures encode important properties of graphs in many application domains. For example, consider the series of phenolic acids illustrated in Fig. 17.10. These represent a family of organic compounds with similar chemical properties. Many complex variations of this family act as signaling molecules and agents of defense in plants. The properties of phenolic acids are a direct result of the presence of two frequent substructures, corresponding to the carboxyl group and phenol group, respectively. These groups are illustrated in Fig. 17.10 as well. The relevance of such substructural properties is not restricted to the chemical domain. This is the reason that frequent substructures are often used in the intermediate stages of many graph mining applications such as clustering and classification.

The definition of a frequent subgraph is identical to the case of association pattern mining, except that a subgraph relationship is used to count the support rather than a subset relationship. Many well-known frequent substructure mining algorithms are based on the enumeration tree principle discussed in Chap. 4. The simplest of these methods is based on the *Apriori* algorithm. This algorithm is discussed in detail in Fig. 4.2 of Chap. 4. The *Apriori* algorithm uses joins to create candidate patterns of size $(k + 1)$ from frequent patterns of size $k$. However, because of the greater complexity of graph-structured data, the join between a pair of graphs may not result in a unique solution. For example, candidate frequent patterns can be generated by either *node extensions* or *edge extensions*. Thus, the main difference between these two variations is in terms of how frequent substructures of size $k$ are defined and joined together to create candidate structures of size $(k + 1)$. *The "size" of a subgraph may refer to either the number of nodes in it, or the number of edges in it depending on whether node extensions or edge extensions are used.* Therefore, the following will describe the *Apriori*-based algorithm in a general way without specifically discussing

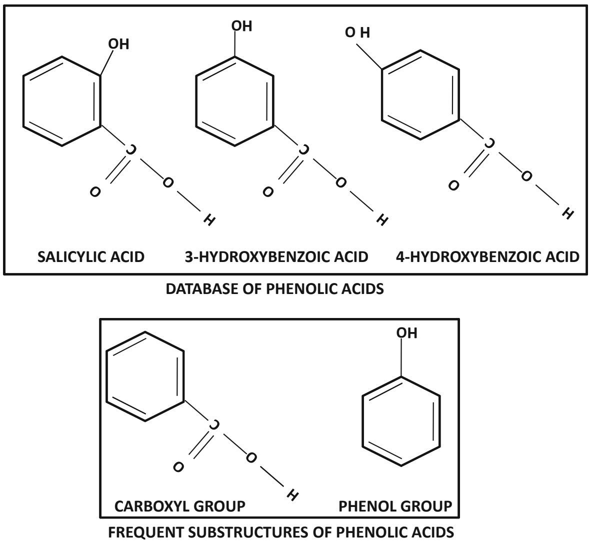


Figure 17.10: Examples of frequent substructures in a database of phenolic acids

either node extensions or edge extensions. Subsequently, the precise changes required to enable these two specific variations will be discussed.

The overall algorithm for frequent subgraph mining is illustrated in Fig. 17.11. The input to the algorithm is the graph database $\mathcal{G} = \{G_1 \ldots G_n\}$ and a minimum support value *minsup*. The basic algorithm structure is similar to that of the *Apriori* algorithm, discussed in Fig. 4.2 of Chap. 4. A levelwise algorithm is used, in which candidate subgraphs $\mathcal{C}_{k+1}$ of size $(k+1)$ are generated by using joins on graph pairs from the set of frequent subgraphs $\mathcal{F}_k$ of size $k$. As discussed earlier, the size of a subgraph may refer to either its nodes or edges, depending on the specific algorithm used. The two graphs need to be matching in a subgraph of size $(k-1)$ for a join to be successfully performed. The resulting candidate subgraph will be of size $(k+1)$. Therefore, one of the important steps of join processing, is determining whether two graphs share a subgraph of size $(k-1)$ in common. The matching algorithms discussed in Sect. 17.2 can be used for this purpose. In some applications, where node labels are distinct and isomorphism is not an issue, this step can be performed very efficiently. On the other hand, for large graphs that have many repeating node labels, this step is slow because of isomorphism.

After the pairs of matching graphs have been identified, joins are performed on them in order to generate the candidates $\mathcal{C}_{k+1}$ of size $(k+1)$. The different node-based and edge-based variations in the methods for performing joins will be described later. Furthermore, the *Apriori* pruning trick is used. Candidates in $\mathcal{C}_{k+1}$ that are such that any of their $k$-subgraphs do not exist in $\mathcal{F}_k$ are pruned. For each remaining candidate subgraph, the support is computed with respect to the graph database $\mathcal{G}$. The subgraph isomorphism algorithm discussed in Sect. 17.2 needs to be used for computing the support. All candidates in $\mathcal{C}_{k+1}$ that meet the minimum support requirement are retained in $\mathcal{F}_{k+1}$. The procedure is repeated iteratively until an empty set $\mathcal{F}_{k+1}$ is generated. At this point, the algorithm terminates, and the set of frequent subgraphs in $\cup_{i=1}^{k} \mathcal{F}_i$ is reported. Next, the two different ways of defining the size $k$ of a graph, corresponding to node- and edge-based joins, will be described.

```
**Algorithm** *GraphApriori*(Graph Database: $\mathcal{G}$,
            Minimum Support: *minsup*);
**begin**
  $\mathcal{F}_1$ = { All Frequent singleton graphs };
  $k = 1$;
  **while** $\mathcal{F}_k$ is not empty **do begin**
    Generate $\mathcal{C}_{k+1}$ by joining pairs of graphs in $\mathcal{F}_k$ that
        share a subgraph of size $(k-1)$ in common;
    Prune subgraphs from $\mathcal{C}_{k+1}$ that violate downward closure;
    Determine $\mathcal{F}_{k+1}$ by support counting on $(\mathcal{C}_{k+1}, \mathcal{G})$ and retaining
            subgraphs from $\mathcal{C}_{k+1}$ with support at least *minsup*;
    $k = k + 1$;
  **end**;
  **return**($\cup_{i=1}^{k} \mathcal{F}_i$);
**end**
```

Figure 17.11: The basic frequent subgraph discovery algorithm is related to the *Apriori* algorithm. The reader is encouraged to compare this pseudocode with the *Apriori* algorithm described in Fig. 4.2 of Chap. 4.

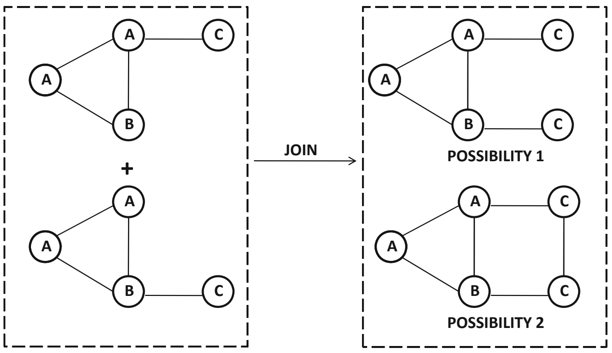


Figure 17.12: Candidates generated using node-based join of two graphs

### 17.4.1 Node-Based Join Growth

In the case of node-based joins, the size of a frequent subgraph in $\mathcal{F}_k$ refers to the number of *nodes* $k$ in it. The singleton graphs in $\mathcal{F}_1$ contain a single node. These are node labels that are present in at least *minsup* graphs in the graph database $\mathcal{G}$. For two graphs from $\mathcal{F}_k$ to be joined, *a matching subgraph with* $(k-1)$ *nodes* must exist between the two graphs. This matching subgraph is also referred to as the *core*. When two $k$-subgraphs with $(k-1)$ common nodes are joined to create a candidate with $(k+1)$ nodes, an ambiguity exists, as to whether or not an edge exists between the two non-matching nodes. Therefore, two possible graphs are generated, depending on whether or not an edge exists between the nodes that are not common between the two. An example of the two possibilities for generating candidate subgraphs is illustrated in Fig. 17.12. While this chapter does not assume that *edge* labels are associated with graphs, the number of possible joins will be even larger when labels are associated with edges. This is because each possible edge label must be associated with the newly created edge. This will result in a larger number of candidates. Furthermore, in cases where there are isomorphic matchings of size $(k-1)$ between the two frequent subgraphs, candidates may need to be generated for each such mapping (see Exercise 8). Thus, all possible $(k-1)$ common subgraphs need to be discovered between a pair of graphs, in order to generate the candidates. Thus, the explosion in the number of candidate patterns is usually more significant in the case of frequent subgraph discovery, than in the case of frequent pattern discovery.

### 17.4.2 Edge-Based Join Growth

In the case of edge-based joins, the size of a frequent subgraph in $\mathcal{F}_k$ refers to the number of *edges* $k$ in it. The singleton graphs in $\mathcal{F}_1$ contain a single edge. These correspond to edges between specific node labels that are present in at least *minsup* graphs in the database $\mathcal{G}$. In order for two graphs from $\mathcal{F}_k$ to be joined, *a matching subgraph with* $(k-1)$ *edges* needs to be present in the two graphs. The resulting candidate will contain exactly $(k+1)$ edges. Interestingly, the number of *nodes* in the candidate may not necessarily be greater than that in the individual subgraphs that are joined. In Fig. 17.13, the two possible candidates that are constructed using edge-based joins are illustrated. Note that one of the generated candidates has the same number of nodes as the original pair of graphs. As in the case of node-based joins, one needs to account for isomorphism in the process of candidate generation. Edge-based join growth tends to generate fewer candidates in total and is therefore generally more efficient. The bibliographic notes contain pointers to more details about these methods.

### 17.4.3 Frequent Pattern Mining to Graph Pattern Mining

The similarity between the aforementioned approach and *Apriori* is quite striking. The join-based growth strategy can also be generalized to an enumeration tree-like strategy. However, the analogous candidate tree can be generated in two different ways, corresponding to node- and edge-based extensions, respectively. Furthermore, tree growth is more complex because of isomorphism. *GraphApriori* uses a breadth-first candidate-tree generation approach as in all *Apriori*-like methods. It is also possible to use other strategies, such as depth-first methods, to grow the tree of candidates. As discussed in Chap. 4, almost all frequent pattern mining algorithms, including[1] *Apriori* and *FP-growth*, should be considered enumeration-

[1]See the discussion in Sect. 4.4.4.5 of Chap. 4.

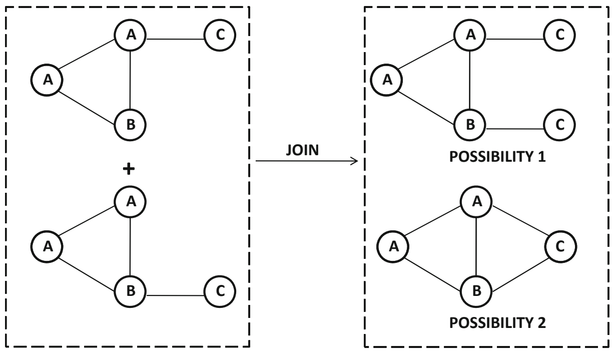


Figure 17.13: Candidates generated using edge-based join of two graphs

tree methods. Therefore, the broader principles of these algorithms can also be generalized to the growth of the candidate tree in graphs. The bibliographic notes contain pointers to these methods.

## 17.5 Graph Clustering

The graph clustering problem partitions a database of $n$ graphs, denoted by $G_1 \dots G_n$, into groups. Graph clustering methods are either distance-based or frequent substructure-based. Distance-based methods are more effective for smaller graphs, in which distances can be computed robustly and efficiently. Frequent substructure-based methods are appropriate for larger graphs where distance computations become qualitatively and computationally impractical.

### 17.5.1 Distance-Based Methods

The design of distance functions is particularly important for virtually every complex data type because of their applicability to clustering methods, such as $k$-medoids and spectral methods, that are dependent only on the design of the distance function. Virtually all the complex data types discussed in Chaps. 13–16 use this general methodology for clustering. This is the reason that distance function design is usually the most fundamental problem that needs to be addressed in every data domain. Sections 17.2 and 17.3 of this chapter have discussed methods for distance computation in graphs. After a distance function has been designed, the following two methods can be used:

1. The $k$-medoids method introduced in Sect. 6.3.4 in Chap. 6 uses a representative-based approach, in which the distances of data objects to their closest representatives are used to perform the clustering. A set of $k$ representatives is used, and data objects are assigned to their closest representatives by using an appropriately designed distance function. The set of $k$ representatives is progressively optimized by using a hill-climbing approach, in which the representatives are iteratively swapped with other data objects in order to improve the clustering objective function value. The reader is referred to Chap. 6 for details of the $k$-medoids algorithm. A key property of this algorithm is that the computations are not dependent on the nature of the data type after the distance function has been defined.

2. A second commonly-used methodology is that of spectral methods. In this case, the individual graph objects are used to construct a single large neighborhood graph. The latter graph is a higher level similarity graph, in which each node corresponds to one of the (smaller) graph objects from the original database and the weight of the edge is equal to the similarity between the two objects. As discussed in Sect. 6.7 of Chap. 6, distances can be converted to similarity values with the use of a kernel transformation. Each node is connected to its $k$-nearest neighbors with an undirected edge. Thus, the problem of clustering graph *objects* is transformed to the problem of clustering *nodes* in a single large graph. This problem is discussed briefly in Sect. 6.7 of Chap. 6, and in greater detail in Sect. 19.3 of Chap. 19. Any of the network clustering or community detection algorithms can be used to cluster the nodes, although spectral methods are used quite commonly. After the node clusters have been determined, they are mapped back to clusters of graph objects.

The aforementioned methods do not work very well when the individual graph objects are large because of two reasons. It is generally computationally expensive to compute distances between large graph objects. Graph distance functions, such as matching-based methods, have a complexity that increases exponentially with graph object size. The *effectiveness* of such methods also drops sharply with increasing graph size. This is because the graphs may be similar only in some portions that repeat frequently. The rare portions of the graphs may be unique to the specific graph at hand. In fact, many small substructures may be repeated across the two graphs. Therefore, a matching-based distance function may not be able to properly compare the key features of the different graphs. One possibility is to use a substructure-based distance function, as discussed in Sect. 17.3.1. A more direct approach is to use frequent substructure-based methods.

## 17.5.2 Frequent Substructure-Based Methods

These methods extract frequent subgraphs from the data and use their membership in input graphs to determine clusters. The basic premise is that the frequent subgraphs are indicative of cluster membership because of their propensity to define application-specific properties. For example, in an organic chemistry application, a benzene ring (illustrated as a subgraph of Fig. 17.1a) is a frequently occurring substructure that is indicative of specific chemical properties of the compound. In an XML application, a frequent substructure corresponds to important structural relationships between entities. Therefore, the membership of such substructures in graphs is highly indicative of similarity and cluster membership. Interestingly, frequent pattern mining algorithms are also used in multidimensional clustering. An example is the *CLIQUE* algorithm (cf. Sect. 7.4.1 of Chap. 7).

In the following sections, two different methods for graph clustering will be described. The first is a generic transformational approach that can be used to apply text clustering methods to the graph domain. The second is a more direct iterative approach of relating the graph clusters to their frequent substructures.

### 17.5.2.1 Generic Transformational Approach

This approach transforms the graph database to a text-like domain, so that the wide variety of text clustering algorithms may be leveraged. The broad approach may be described as follows:

1. Apply frequent subgraph mining methods discussed in Sect. 17.4 in order to discover frequent subgraph patterns in the underlying graphs. Select a subset of subgraphs to

reduce overlap among the different subgraphs. Different algorithms may vary on this step by using only frequent maximal subgraphs, or selecting a subset of graphs that are sufficiently nonoverlapping with one another. Create a new feature $f_i$ for each frequent subgraph $S_i$ that is discovered. Let $d$ be the total number of frequent subgraphs (features). This is the "lexicon" size in terms of which a text-like representation will be constructed.

2. For each graph $G_i$, create a vector-space representation in terms of the features $f_1 \ldots f_d$. Each graph contains the features corresponding to the subgraphs that it contains. The frequency of each feature is the number of occurrences of the corresponding subgraph in the graph $G_i$. It is also possible to use a binary representation by only considering the presence or absence of subgraphs rather than frequency of presence. Use tf-idf normalization on the vector-space representation, as discussed in Chap. 13.

3. Use any of the text-clustering algorithms discussed in Sect. 13.3 in Chap. 13, in order to discover clusters of newly created text objects. Map the text clusters to graph object clusters.

This broader approach of using text-based methods is utilized frequently with many contextual data types. For example, an almost exactly analogous approach is discussed for sequence clustering in Sect. 15.3.3 of Chap. 15. This is because a modified version of frequent pattern mining methods can be defined for most data types. It should be pointed out that, although the substructure-based transformation is discussed here, many of the kernel-based transformations and topological descriptors, discussed earlier in this chapter, may be used as well. For example, the kernel $k$-means algorithm can be used in conjunction with the graph kernels discussed in this chapter.

#### 17.5.2.2 XProj: Direct Clustering with Frequent Subgraph Discovery

The *XProj* algorithm derives its name from the fact that it was originally proposed for XML graphs, and a substructure can be viewed as a PROJection of the graph. Nevertheless, the approach is not specific to XML structures, and it can be applied to any other graph domain, such as chemical compounds. The *XProj* algorithm uses the substructure discovery process as an important subroutine, and different applications may use different substructure discovery methods, depending on the data domain. Therefore, the following will provide a generic description of the *XProj* algorithm for graph clustering, although the substructure discovery process may be implemented in an application-specific way. Because the algorithm uses the frequent substructures for the clustering process, an additional input to the algorithm is the minimum support *minsup*. Another input to the algorithm is the size $l$ of the frequent substructures mined. The size of the frequent substructures is fixed in order to ensure robust computation of similarity. These are user-defined parameters that can be tuned to obtain the most effective results.

The algorithm can be viewed as a representative approach similar to $k$-medoids, except that *each representative is a set of frequent substructures*. These represent the *localized substructures* of each group. The use of frequent-substructures as the representatives, instead of the original graphs, is crucial. This is because distances cannot be computed effectively between pairs of graphs, when the sizes of the graphs are larger. On the other hand, the membership of frequent substructures provides a more intuitive way of computing similarity. It should be pointed out that, unlike transformational methods, the frequent substructures

```
Algorithm XProj(Graph Database: $\mathcal{G}$, Minimum Support: minsup
          Structural Size: $l$, Number of Clusters: $k$ )
begin
  Initialize clusters $\mathcal{C}_1 \ldots \mathcal{C}_k$ randomly;
  Compute frequent substructure sets $\mathcal{F}_1 \ldots \mathcal{F}_k$ from $\mathcal{C}_1 \ldots \mathcal{C}_k$;
  repeat
    Assign each graph $G_j \in \mathcal{G}$ to the cluster $\mathcal{C}_i$ for which the former's
      similarity to $\mathcal{F}_i$ is the largest $\forall i \in \{1 \ldots k\}$;
    Compute frequent substructure set $\mathcal{F}_i$ from $\mathcal{C}_i$ for each $i \in \{1 \ldots k\}$;
  until convergence;
end
```

Figure 17.14: The frequent subgraph-based clustering algorithm (high level description)

are local to each cluster, and are therefore better optimized. This is the main advantage of this approach over a generic transformational approach.

There are a total of $k$ such frequent substructure sets $\mathcal{F}_1 \ldots \mathcal{F}_k$, and the graph database is partitioned into $k$ groups around these localized representatives. The algorithm is initialized with a random partition of the database $\mathcal{G}$ into $k$ clusters. These $k$ clusters are denoted by $\mathcal{C}_1 \ldots \mathcal{C}_k$. The frequent substructures $\mathcal{F}_i$ of each of these clusters $\mathcal{C}_i$ can be determined using any frequent substructure discovery algorithm. Subsequently, each graph in $G_j \in \mathcal{G}$ is assigned to one of the representative sets $\mathcal{F}_i$ based on the similarity of $G_j$ to each representative set $\mathcal{F}_i$. The details of the similarity computation will be discussed later. This process is repeated iteratively, so that the representative set $\mathcal{F}_i$ is generated from cluster $\mathcal{C}_i$, and the cluster $\mathcal{C}_i$ is generated from the frequent set $\mathcal{F}_i$. The process is repeated, until the change in the average similarity of each graph $G_j$ to its assigned representative set $\mathcal{F}_i$ is no larger than a user-defined threshold. At this point, the algorithm is assumed to have converged, and it terminates. The overall algorithm is illustrated in Fig. 17.14.

It remains to be described how the similarity between a graph $G_j$ and a representative set $\mathcal{F}_i$ is computed. The similarity between $G_j$ and $\mathcal{F}_i$ is computed with the use of a coverage criterion. The similarity between $G_j$ and $\mathcal{F}_i$ is equal to the fraction of frequent substructures in $\mathcal{F}_i$ that are a subgraph of $G_j$.

A major computational challenge is that the determination of frequent substructures in $\mathcal{F}_i$ may be too expensive. Furthermore, there may be a large number of frequent substructures in $\mathcal{F}_i$ that are highly overlapping with one another. To address these issues, the *XProj* algorithm proposes a number of optimizations. The first optimization is that the frequent substructures do not need to be determined exactly. An approximate algorithm for frequent substructure mining is designed. The second optimization is that only a subset of nonoverlapping substructures of length $l$ are included in the sets $\mathcal{F}_i$. The details of these optimizations may be found in pointers discussed in the bibliographic notes.

## 17.6 Graph Classification

It is assumed that a set of $n$ graphs $G_1 \ldots G_n$ is available, but only a subset of these graphs is labeled. Among these, the first $n_t \leq n$ graphs are labeled, and the remaining $(n - n_t)$ graphs are unlabeled. The labels are drawn from $\{1 \ldots k\}$. It is desired to use the labels on the training graphs to infer the labels of unlabeled graphs.

### 17.6.1 Distance-Based Methods

Distance-based methods are most appropriate when the sizes of the underlying graphs are small, and the distances can be computed efficiently. Nearest neighbor methods and collective classification methods are two of the distance-based methods commonly used for classification. The latter method is a transductive semi-supervised method, in which the training and test instances need to be available at the same time for the classification process. These methods are described in detail below:

1. *Nearest neighbor methods:* For each test instance, the $k$-nearest neighbors are determined. The dominant label from these nearest neighbors is reported as the relevant label. The nearest neighbor method for multidimensional data is described in detail in Sect. 10.8 of Chap. 10. The only modification to the method is the use of a different distance function, suited for the graph data type.

2. *Graph-based methods:* This is a semi-supervised method, discussed in Sect. 11.6.3 of Chap. 11. In graph-based methods, a higher level *neighborhood graph* is constructed from the training and test graph objects. It is important not to confuse the notion of a neighborhood graph with that of the original graph objects. The original graph objects correspond to nodes in the neighborhood graph. Each node is connected to its $k$ nearest neighbor objects based on the distance values. This results in a graph containing both labeled and unlabeled nodes. This is the collective classification problem, for which various algorithms are described in Sect. 19.4 of Chap. 19. Collective classification algorithms can be used to derive labels of the nodes in the neighborhood graphs. These derived labels can then be mapped back to the unlabeled graph objects.

Distance-based methods are generally effective when the underlying graph objects are small. For larger graph objects, the computation of distances becomes too expensive. Furthermore, distance computations no longer remain effective from an accuracy perspective, when multiple common substructures are present in the two graphs.

### 17.6.2 Frequent Substructure-Based Methods

Pattern-based methods extract frequent subgraphs from the data, and use their membership in different graphs, in order to build classification models. As in the case of clustering, the main assumption is that the frequently occurring portions of graphs can related to application-specific properties of the graphs. For example, the phenolic acids of Fig. 17.10 are characterized by the two frequent substructures corresponding to the carboxyl group and the phenol group. These substructures therefore characterize important properties of a *family* or a *class* of compounds. This is generally true across many different applications beyond the chemical domain. As discussed in Sect. 10.4 of Chap. 10, frequent patterns are often used for rule-based classification, even in the "flat" multidimensional domain. As in the case of clustering, either a generic transformational approach or a more direct rule-based method can be used.

#### 17.6.2.1 Generic Transformational Approach

This approach is generally similar to the transformational approach discussed in the previous section on clustering. However, there are a few differences that account for the impact of supervision. The broad approach may be described as follows:

1. Apply frequent subgraph mining methods discussed in Sect. 17.4 to discover frequent subgraph patterns in the underlying graphs. Select a subset of subgraphs to reduce overlap among the different subgraphs. For example, feature selection algorithms that minimize redundancy and maximize the relevance of the features may be used. Such feature selection algorithms are discussed in Sect. 10.2 of Chap. 10. Let $d$ be the total number of frequent subgraphs (features). This is the "lexicon" size in terms of which a text-like representation will be constructed.

2. For each graph $G_i$, create a vector-space representation in terms of the $d$ features found. Each graph contains the features corresponding to the subgraphs that it contains. The frequency of each feature is equal to the number of occurrences of the corresponding subgraph in graph $G_i$. It is also possible to use a binary representation by only considering presence or absence of subgraphs, rather than frequency of presence. Use tf-idf normalization on the vector-space representation, as discussed in Chap. 13.

3. Select any text classification algorithm discussed in Sect. 13.5 of Chap. 13 to build a classification model. Use the model to classify test instances.

This approach provides a flexible framework. After the transformation has been performed, a wide variety of algorithms may be used. It also allows the use of different types of supervised feature selection methods to ensure that the most discriminative structures are used for classification.

#### 17.6.2.2 XRules: A Rule-Based Approach

The *XRules* method was proposed in the context of XML data, but it can be used in the context of any graph database. This is a rule-based approach that relates frequent substructures to the different classes. The training phase contains three steps:

1. In the first phase, frequent substructures with sufficient support and confidence are determined. Each rule is of the form:

   $$F_g \Rightarrow c$$

   The notation $F_g$ denotes a frequent substructure, and $c$ is a class label. Many other measures can be used to quantify the strength of the rule instead of the confidence. Examples include the likelihood ratio, or the cost-weighted confidence in the rare class scenario. The likelihood ratio of $F_g \Rightarrow c$ is the ratio of the fractional support of $F_g$ in the examples containing $c$, to the fractional support of $F_g$ in examples not containing $c$. A likelihood ratio greater than one indicates that the rule is highly likely to belong to a particular class. The generic term for these different ways of measuring the class-specific relevance is the *rule strength*.

2. In the second phase, the rules are ordered and pruned. The rules are ordered by decreasing strength. Statistical thresholds on the rule strength may be used for pruning rules with low strength. This yields a compact set $\mathcal{R}$ of ordered rules that are used for classification.

3. In the final phase, a default class is set that can be used to classify test instances not covered by any rule in $\mathcal{R}$. The default class is set to the dominant class of the set of training instances not covered by rule set $\mathcal{R}$. A graph is covered by a rule if the

left-hand side of the rule is a substructure of the graph. In the event that all training instances are covered by rule set $\mathcal{R}$, then the default class is set to the dominant class in the entire training data. In cases where classes are associated with costs, the cost-sensitive weight is used in determining the majority class.

After the training model has been constructed, it can be used for classification as follows. For a given test graph $G$, the rules that are fired by $G$ are determined. If no rules are fired, then the default class is reported. Let $\mathcal{R}_c(G)$ be the set of rules fired by $G$. Note that these different rules may not yield the same prediction for $G$. Therefore, the conflicting predictions of different rules need to be combined meaningfully. The different criteria that are used to combine the predictions are as follows:

1. *Average strength:* The average strength of the rules predicting each class are determined. The class with the average highest strength is reported.

2. *Best rule:* The top rule is determined on the basis of the priority order discussed earlier. The class label of this rule is reported.

3. *Top-$k$ average strength:* This can be considered a combination of the previous two methods. The average strength of the top-$k$ rules of each class is used to determine the predicted label.

The *XRules* procedure uses an efficient procedure for frequent substructure discovery, and many other variations for rule quantification. Refer to the bibliographic notes.

### 17.6.3 Kernel SVMs

Kernel SVMs can construct classifiers with the use of kernel similarity between training and test instances. As discussed in Sect. 10.6.4 of Chap. 10, kernel SVMs do not actually need the feature representation of the data, as long as the kernel-based similarity $K(G_i, G_j)$ between any pair of graph objects is available. Therefore, the approach is agnostic to the specific data type that is used. The different kinds of graph kernels are discussed in Sect. 17.3.3 of this chapter. Any of these kernels can be used in conjunction with the *SVM* approach. Refer to Sect. 10.6.4 of Chap. 10 for details on how kernels may be used in conjunction with SVM classifiers.

## 17.7 Summary

This chapter studies the problem of mining graph data sets. Graph data are a challenging domain for analysis, because of the difficulty in matching two graphs when there are repetitions in the underlying labels. This is referred to as graph isomorphism. Most methods for graph matching require exponential time in the worst case. The MCG between a pair of graphs can be used to define distance measures between graphs. The edit-distance measure also uses an algorithm that is closely related to the MCG algorithm. Because of the complexity of matching algorithms in graphs, a different approach is to transform the graph database into a simpler text-like representation, in terms of which distance functions are defined. An important class of graph distance functions is graph kernels. They can be used for clustering and classification.

The frequent substructure discovery algorithm is an important building block because it can be leveraged for other graph mining problems such as clustering and classification.

The *Apriori*-like algorithms use either a node-growth strategy, or an edge-growth strategy in order to generate the candidates and corresponding frequent substructures. Most of the clustering and classification algorithms for graph data are based either on distances, or on frequent substructures. The distance-based methods include the $k$-medoids and spectral methods for clustering. For classification, the distance-based methods include either the $k$-nearest neighbor method or graph-based semi-supervised methods. Kernel-based SVM can also be considered specialized distance-based methods, in which SVMs are leveraged in conjunction with the similarity between data objects.

Frequent substructure-based methods are used frequently for graph clustering and classification. A generic approach is to transform the graphs into a new feature representation that is similar to text data. Any of the text clustering or classification algorithms can be applied to this representation. Another approach is to directly mine the frequent substructures and use them as representative sets of clusters, or antecedents of discriminative rules. The *XProj* and *XRules* algorithms are based on this principle.

## 17.8 Bibliographic Notes

The problem of graph matching is addressed in surveys in [26]. The Ullman algorithm for graph matching was proposed in [164]. Two other well known methods for graph-matching are *VF2* [162] and *QuickSI* [163]. Other approximate matching methods are discussed in [313, 314, 521]. The proof of NP-hardness of the graph matching problem may be found in [221, 164]. The use of the MCG for defining distance functions was studied in [120]. The relationship between the graph-edit distance and the maximum common subgraph problem is studied in detail in [119]. The graph edit-distance algorithm discussed in the chapter is a simplification of the algorithm presented in [384]. A number of fast algorithms for computing the graph edit distance are discussed in [409]. The problem of learning edit costs is studied in [408]. The survey by Bunke in [26] also discusses methods for computing the graph edit costs. A description of the use of topological descriptors in drug-design may be found in [236]. The random walk kernel is discussed in [225, 298], and the shortest-path kernel is discussed in [103]. The work in [225] also provides a generic discussion on graph kernels. The work in [42] shows that frequent substructure-based similarity computation can provide robust results in data mining applications.

The node-growth strategy for frequent subgraph mining was proposed by Inokuchi, Washio, an Motoda [282]. The edge-growth strategy was proposed by Kuramochi and Karypis [331]. The *gSpan* algorithm was proposed by Yan and Han [519] and uses a depth-first approach to build the candidate tree of graph patterns. A method that uses the vertical representation for graph pattern mining is discussed in [276]. The problem of mining frequent trees in a forest was addressed in [536]. Surveys on graph clustering and classification may be found in [26]. The *XProj* algorithm is discussed in [42], and the *XRules* algorithm is discussed in [540]. Methods for kernel SVM-based classification are discussed in the graph classification chapter by Tsuda in [26].

## 17.9 Exercises

**1.** Consider two graphs that are cliques containing an even number $2 \cdot n$ nodes. Let exactly half the nodes in each graph belong to labels $A$ and $B$. What are the total number of isomorphic matchings between the two graphs?

2. Consider two graphs containing $2 \cdot n$ nodes and $n$ distinct labels, each of which occurs twice. What is the maximum number of isomorphic matchings between the two graphs?

3. Implement the basic algorithm for subgraph isomorphism with no pruning optimizations. Test it by trying to match pairs of randomly generated graphs, containing a varying number of nodes. How does the running time vary with the size of the graph?

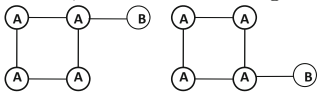
4. Compute the Morgan indices of order 1 and 2, for each node of the *acetaminophen* graph of Fig. 17.1. How does the Morgan index vary with the labels (corresponding to chemical elements)?

5. Write a computer program to compute each of the topological descriptors of a graph discussed in this chapter.

6. Write a computer program to execute the node-based candidate growth for frequent subgraph discovery. Refer to the bibliographic notes, if needed, for the paper describing specific details of the algorithm.

7. Write a computer program to execute the edge-based candidate growth for frequent subgraph discovery. Refer to the bibliographic notes for the paper describing specific details of the algorithm.

8. Show the different node-based joins that can be performed between the two graphs below, while accounting for isomorphism.



9. Show the different edge-based joins that can performed between the two graphs of Exercise 8, while accounting for isomorphism.

10. Determine the maximum number of candidates that can be generated with node-based join growth using a single pair of graphs, while accounting for isomorphism. Assume that the matching core of these graphs is a cycle of size $k$. What conditions in the core of the joined portion result in this scenario?

11. Discuss how the node-based growth and edge-based growth strategies translate into a candidate tree structure that is analogous to the enumeration tree in frequent pattern mining.

12. Implement a computer program to construct a text-like representation for a database of graphs, as discussed in the chapter. Use any feature selection approach of your choice of minimize redundancy. Implement a $k$-means clustering algorithm with this representation.

13. Repeat Exercise 12 for the classification problem. Use a naive Bayes classifier, as discussed in Chapter 10, for the final classification step and an appropriately chosen supervised feature selection method from the same chapter.

14. What changes would be require in the subgraph isomorphism algorithm for cases in which the query graph is disconnected?