Chapter 2
Frequent Pattern Mining Algorithms: A Survey

Charu C. Aggarwal, Mansurul A. Bhuiyan and Mohammad Al Hasan

Abstract This chapter will provide a detailed survey of frequent pattern mining algorithms. A wide variety of algorithms will be covered starting from *Apriori*. Many algorithms such as *Eclat*, *TreeProjection*, and *FP-growth* will be discussed. In addition a discussion of several maximal and closed frequent pattern mining algorithms will be provided. Thus, this chapter will provide one of most detailed surveys of frequent pattern mining algorithms available in the literature.

Keywords Frequent pattern mining algorithms · *Apriori* · *TreeProjection* · *FP-growth*

1 Introduction

In data mining, frequent pattern mining (FPM) is one of the most intensively investigated problems in terms of computational and algorithmic development. Over the last two decades, numerous algorithms have been proposed to solve frequent pattern mining or some of its variants, and the interest in this problem still persists [45, 75]. Different frameworks have been defined for frequent pattern mining. The most common one is the support-based framework, in which itemsets with frequency above a given threshold are found. However, such itemsets may sometimes not represent interesting positive *correlations* between items because they do not normalize for the absolute frequencies of the items. Consequently, alternative measures for interestingness have been defined in the literature [7, 11, 16, 63]. This chapter will focus on the support-based framework because the algorithms based on the interestingness

C. C. Aggarwal (✉)
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
e-mail: charu@us.ibm.com

M. A. Bhuiyan · M. A. Hasan
Indiana University–Purdue University, Indianapolis, IN, USA
e-mail: mbhuiyan@cs.iupui.edu

M. A. Hasan
e-mail: alhasan@cs.iupui.edu

```
Algorithm Baseline Mining(Database: $\mathcal{T}$, Minimum Support: $s$)
  begin
    $\mathcal{FP} = \{\}$;
    Insert length-one frequent pattern in $\mathcal{FP}$
    until all frequent patterns in $\mathcal{FP}$ are explored do
    begin
      Generate a candidate pattern $P$ from one (or more) frequent
              pattern(s) in $\mathcal{FP}$
      if support($P, \mathcal{T}$) $\geq s$
        Add $P$ to frequent pattern set $\mathcal{FP}$;
    end
  end
```

Fig. 2.1 A generic frequent pattern mining algorithm

framework are provided in a different chapter. Surveys on frequent pattern mining may be found in [26, 33].

One of the main reasons for the high level of interest in frequent pattern mining algorithms is due to the computational challenge of the task. Even for a moderate sized dataset, the search space of FPM is enormous, which is exponential to the length of the transactions in the dataset. This naturally creates challenges for itemset generation, when the support levels are low. In fact, in most practical scenarios, the support levels at which one can mine the corresponding itemsets are limited (bounded below) by the memory and computational constraints. Therefore, it is critical to be able to perform the analysis in a space- and time-efficient way. During the first few years of research in this area, the primary focus of work was to find FPM algorithms with better computational efficiency.

Several classes of algorithms have been developed for frequent pattern mining, many of which are closely related to one another. In fact, the execution tree of all the algorithms is mostly different in terms of the order in which the patterns are explored, and whether the counting work done for different candidates is independent of one another. To explain this point, we introduce a primitive "baseline" algorithm that forms the heart of most frequent pattern mining algorithms.

Figure 2.1 presents the pseudocode for a very simple "baseline" frequent pattern mining algorithm. The algorithm takes the transaction database $\mathcal{T}$ and a user-defined support value s as input. It first populates all length-one frequent patterns in a frequent pattern data-store, $\mathcal{FP}$. Then it generates a candidate pattern and computes its support in the database. If the support of the candidate pattern is equal or higher than the minimum support threshold the pattern is stored in $\mathcal{FP}$. The process continues until all the frequent patterns from the database are found.

In the aforementioned algorithm, candidate patterns are generated from the previously generated frequent patterns. Then, the transaction database is used to determine which of the candidates are truly frequent patterns. The key issues of computational efficiency arise in terms of generating the candidate patterns in an orderly and carefully designed fashion, pruning irrelevant and duplicate candidates, and using well chosen tricks to minimize the work in counting the candidates. Clearly, the effectiveness of these different strategies depend on each other. For example, the effectiveness of a pruning strategy may be dependent on the order of exploration of the candidates (level-wise vs. depth first), and the effectiveness of counting is also dependent on the order of exploration because the work done for counting at the higher levels (shorter itemsets) can be reused at the lower levels (longer itemsets) with certain strategies, such as those explored in *TreeProjection* and *FP-growth*. Surprising as it might seem, virtually all frequent pattern mining algorithms can be considered complex variations of this simple baseline pseudocode. The major challenge of all of these methods is that the number of frequent patterns and candidate patterns can sometimes be large. This is a fundamental problem of frequent pattern mining although it is possible to speed up the counting of the different candidate patterns with the use of various tricks such as database projections. An analysis on the number of candidate patterns may be found in [25].

The candidate generation process of the earliest algorithms used joins. The original *Apriori* algorithm belongs to this category [1]. Although *Apriori* is presented as a join-based algorithm, it can be shown that the algorithm is a breadth first exploration of a structured arrangement of the itemsets, known as a *lexicographic tree* or *enumeration tree*. Therefore, later classes of algorithms explicitly discuss tree-based enumeration [4, 5]. The algorithms assume a lexicographic tree (or enumeration tree) of candidate patterns and explore the tree using breadth-first or depth-first strategies. The use of the enumeration tree forms the basis for understanding search space decomposition, as in the case of the *TreeProjection* algorithm [5]. The enumeration tree concept is very useful because it provides an understanding of how the search space of candidate patterns may be explored in a systematic and non-redundant way. Frequent pattern mining algorithms typically need to evaluate the support of frequent portions of the enumeration tree, and also rule out an additional layer of infrequent extensions of the frequent nodes in the enumeration tree. This makes the candidate space of all frequent pattern mining algorithms virtually invariant unless one is interested in particular types of patterns such as maximal patterns.

The enumeration tree is defined on the prefixes of frequent itemsets, and will be introduced later in this chapter. Later algorithms such as *FP-growth* perform suffix-based recursive exploration of the search space. In other words, the frequent patterns with a particular pattern as a suffix are explored at one time. This is because *FP-growth* uses the opposite item ordering convention as most enumeration tree algorithms though the recursive exploration order of *FP-growth* is similar to an enumeration tree.

Note that all classes of algorithms, implicitly or explicitly, explore the search space of patterns defined by an enumeration tree of frequent patterns with different strategies such as joins, prefix-based depth-first exploration, or suffix-based depth-first exploration. However, there are significant differences in terms of the order in which the search space is explored, the pruning methods used, and how the counting is performed. In particular, certain projection-based methods help in reusing the counting work for k-itemsets for $(k + 1)$-itemsets with the use of the notion of projected databases. Many algorithms such as *TreeProjection* and *FP-growth* are able to achieve this goal.

Table 2.1 Toy transaction database and frequent items of each transaction for a minimum support of 3

tid	Items	Sorted frequent items
2	a,b,c,d,f,h	a,b,c,d,f
3	a,f,g	a,f
4	b,e,f,g	b,f,e
5	a,b,c,d,e,h	a,b,c,d,e

This chapter is organized as follows. The remainder of this chapter discusses notations and definitions relevant to frequent pattern mining. Section 2 discusses join-based algorithms. Section 3 discusses tree-based algorithms. All the algorithms discussed in Sects. 2 and 3 extend prefixes of itemsets to generated frequent patterns. A number of methods that extend suffixes of frequent patterns are discussed in Sect. 4. Variants of frequent pattern mining, such as closed and maximal frequent pattern mining, are discussed in Sect. 5. Other optimized variations of frequent pattern mining algorithms are discussed in Sect. 6. Methods for reducing the number of passes, with the use of sampling and aggregation are proposed in Sect. 7. Finally, Sect. 8 concludes chapter with an overall summary.

1.1 Definitions

In this section, we define several key concepts of frequent pattern mining (FPM) that we will use in the remaining part of the chapter.

Let, $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a transaction database, where each $T_i \in \mathcal{T}, \forall i = \{1 \ldots n\}$ consists of a set of items, say $T_i = \{x_1, x_2, x_3, \ldots x_l\}$. A set $P \subseteq T_i$ is called an itemset. The size of an itemset is defined by the number of items it contains. We will refer an itemset as l-$itemset$ (or l-$pattern$), if its size is l. The number of transactions containing P is referred to as the *support* of P. A pattern P is defined to be frequent if its support is at least equal to the the minimum threshold.

Table 2.1 depicts a toy database with 5 transactions (T_1, T_2 T_3, T_4 and T_5). The second column shows the items in each transaction. In the third column, we show the set of items that are frequent in the corresponding transaction for a minimum support value of 3. For example, the item h in transaction with *tid* value of 2 is an infrequent item with a support value of 2. Therefore, it is not listed in the third column of the corresponding row. Similarly, the pattern $\{a, b\}$ (or, *ab* in abbreviated form) is frequent because it has a support value of 3.

The frequent patterns are often used to generate *association rules*. Consider the rule $X \Rightarrow Y$, where X and Y are sets of items. The confidence of the rule $X \Rightarrow Y$ is the equal to the ratio of the support of $X \cup Y$ to that of the support of X. In other words, it can be viewed as the conditional probability that Y occurs, given that X has occurred. The support of the rule is equal to the support of $X \cup Y$. Association rule-generation is a two-phase process. The first phase determines all the frequent patterns at a given minimum support level. The second phase extracts all the rules from these patterns. The second phase is fairly trivial and with limited sophistication. Therefore, most of the algorithmic work in frequent pattern mining focusses on the

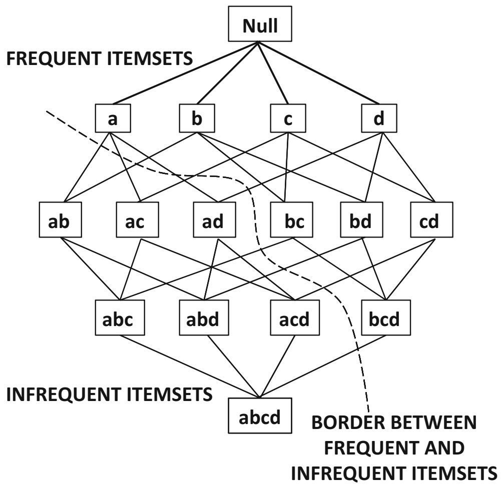

Fig. 2.2 The lattice of itemsets

first phase. This chapter will also focus on the first phase of frequent pattern mining, which is generally considered more important and non-trivial.

Frequent patterns satisfy a *downward closure property*, according to which every subset of a frequent pattern is also frequent. This is because if a pattern P is a subset of a transaction, then every pattern $P' \subseteq P$ will also be a subset of T. Therefore, the support of P' can be no less than that of P. The space of exploration of frequent patterns can be arranged as a lattice, in which every node is one of the 2^d possible itemsets, and an edge represents an immediate subset relationship between these itemsets. An example of a lattice of possible itemsets for a universe of items corresponding to $\{a, b, c, d\}$ is illustrated in Fig. 2.2. The lattice represents the search of frequent patterns, and all frequent pattern mining algorithms must, in one way or another, traverse this lattice to identify the frequent nodes of this lattice. The lattice is separated into a frequent and an infrequent part with the use of a *border*. An example of a border is illustrated in Fig. 2.2. This border must satisfy the downward closure property.

The lattice can be traversed with a variety of strategies such as breadth-first or depth-first methods. Furthermore, *candidate nodes* of the lattice may be generated in many ways, such as using joins, or using lexicographic tree-based extensions. Many of these methods are conceptually equivalent to one another. The following discussion will provide an overview of the different strategies that are commonly used.

2 Join-Based Algorithms

Join-based algorithms generate $(k+1)$-candidates from frequent k-patterns with the use of joins. These candidates are then validated against the transaction database. The *Apriori* method uses joins to create candidates from frequent patterns, and is one of the earliest algorithms for frequent pattern mining.

2.1 Apriori Method

The most basic join-based algorithm is the *Apriori* method [1]. The *Apriori* approach uses a *level-wise* approach in which all frequent itemsets of length k are generated before those of length $(k+1)$. The main observation which is used for the *Apriori* algorithm is that every subset of a frequent pattern is also frequent. Therefore, *candidates* for frequent patterns of length $(k+1)$ can be generated from *known* frequent patterns of length k with the use of joins. A join is defined by pairs of frequent k-patterns that have at least $(k-1)$ items in common. Specifically, consider a frequent pattern $\{i_1, i_2, i_3, i_4\}$ that is frequent, but has not yet been discovered because only itemsets of length 3 have been discovered so far. In this case, because the patterns $\{i_1, i_2, i_3\}$ and $\{i_1, i_2, i_4\}$ are frequent, they will be present in the set $\mathcal{F}_3$ of all frequent patterns with length $k = 3$. Note that this particular pair also has $k - 1 = 2$ items in common. By performing a join on this pair, it is possible to create the *candidate* pattern $\{i_1, i_2, i_3, i_4\}$. This pattern is referred to as a *candidate* because it might *possibly* be frequent, and one most either rule it in or rule it out by support counting. Therefore, this candidate is then *validated* against the transaction database by counting its support. Clearly, the design of an efficient support counting method plays a critical role in the overall efficiency of the process. Furthermore, it is important to note that the same candidate can be produced by joining multiple frequent patterns. For example, one might join $\{i_1, i_2, i_3\}$ and $\{i_2, i_3, i_4\}$ to achieve the same result. Therefore, in order to avoid duplication in candidate generation, two itemsets are joined only whether first $(k-1)$ items are the same, based on a lexicographic ordering imposed on the items. This provides all the $(k+1)$-candidates in a non-redundant way.

It should be pointed out that some candidates can be pruned out in an efficient way, without validating them against the transaction database. For any $(k+1)$-candidates, it is checked whether *all* its k subsets are frequent. Although it is already known that two of its subsets contributing to the join are frequent, it is not known whether its remaining subsets are frequent. If all its subsets are not frequent, then the candidate can be pruned from consideration because of the downward closure property. This is known as the *Apriori* pruning trick. For example, in the previous case, if the itemset $\{i_1, i_3, i_4\}$ does not exist in the set of frequent 3-itemsets which have already been found, then the candidate itemset $\{i_1, i_2, i_3, i_4\}$ can be pruned from consideration with no further computational effort. This greatly speeds up the overall algorithm. The generation of 1-itemsets and 2-itemsets is usually performed in a specialized way with more efficient techniques.

Therefore, the basic *Apriori* algorithm can be described recursively in level-wise fashion. the overall algorithm comprises of three steps that are repeated over and over again, for different values of k, where k is the length of the pattern generated in the current iteration. The four steps are those of (i) generation of candidate patterns $\mathcal{C}_{k+1}$ by using joins on the patterns in $\mathcal{F}_k$, (ii) the pruning of candidates from $\mathcal{C}_{k+1}$, for which all subsets to not lie in $\mathcal{F}_k$, and (iii) the validation of the patterns in $\mathcal{C}_{k+1}$ against the transaction database $\mathcal{T}$, to determine the subset of $\mathcal{C}_{k+1}$ which is truly frequent. The algorithm is terminated, when the set of frequent k-patterns $\mathcal{F}_k$ in a given iteration is empty. The pseudo-code of the overall procedure is presented in Fig. 2.3.

Fig. 2.3 The *Apriori* algorithm

```
Algorithm Apriori(Database: $\mathcal{T}$, Support: $s$)
begin
  Generate frequent 1-patterns and 2-patterns
    using specialized counting methods and
    denote by $\mathcal{F}_1$ and $\mathcal{F}_2$;
  $k := 2$;
  while $\mathcal{F}_k$ is not empty do
  begin
    Generate $\mathcal{C}_{k+1}$ by using joins on $\mathcal{F}_k$;
    Prune $\mathcal{C}_{k+1}$ with Apriori subset pruning trick;
    Generate $\mathcal{F}_{k+1}$ by counting candidates in
      $\mathcal{C}_{k+1}$ with respect to $\mathcal{T}$ at support $s$;
    $k := k + 1$;
  end
  return $\cup_{i=1}^{k} \mathcal{F}_i$;
end
```

The computationally intensive procedure in this case is the counting of the candidates in $\mathcal{C}_{k+1}$ with respect to the transaction database $\mathcal{T}$. Therefore, a number of optimizations and data structures have been proposed in [1] (and also the subsequent literature) to speed up the counting process. The data structure proposed in [1] is that of constructing a *hash-tree* to maintain the candidate patterns. A leaf node of the hash-tree contains a list of itemsets, whereas an interior node contains a hash-table. An itemset is mapped to a leaf node of the tree by defining a path from the root to the leaf node with the use of the hash function. At a node of level i, a hash function is applied to the ith item to decide which branch to follow. The itemsets in the leaf node are stored in sorted order. The tree is constructed recursively in top–down fashion, and a minimum threshold is imposed on the number of candidates in the leaf node.

To perform the counting, all possible k-itemsets which are subsets of a transaction are discovered in a *single* exploration of the hash-tree. To achieve this goal *all possible* paths in the hash tree that could correspond to subsets of the transaction, are followed in recursive fashion, to determine which leaf nodes are relevant to that transaction. After the leaf nodes have been discovered, the itemsets at these leaf nodes that are subsets of that transaction are isolated and their count is incremented. The actual selection of the relevant leaf nodes is performed by recursive traversal as follows. At the root node, all branches are followed such that *any* of the items in the transaction hash to one of branches.At a given interior node, if the ith item of the transaction was last hashed, then all items *following it* in the transaction are hashed to determine the possible children to follow. Thus, by following all these paths, the relevant leaf nodes in the tree are determined. The candidates in the leaf node are stored in sorted order, and can be compared efficiently to the hashed sequence of items in the transaction to determine whether they are relevant. This provides a count of the itemsets relevant to the transaction. This process is repeated for each transaction to determine the final support count for each itemset. It should be pointed out that the reason for using a hash function at the intermediate nodes is to reduce the branching factor of the hash tree. However, if desired, a trie can be used explicitly, in which the degree of a

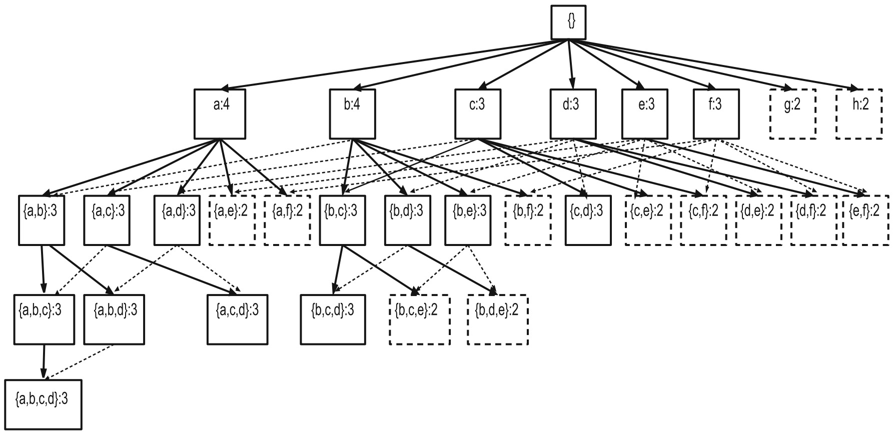

Fig. 2.4 Execution tree of *Apriori* algorithm

node is potentially of the order of the total number of items. An example of such an implementation is provided in [12], and it seems to work quite well. An algorithm that shares some similarities to the *Apriori* method, was independently proposed in [44], and subsequently a combined work was published in [3].

Figure 2.4 illustrates the execution tree of the join-based *Apriori* algorithm over the toy transaction database mentioned in Table 2.1 for minimum support value 3. As mentioned in the pseudocode of *Apriori*, a candidate k-patterns are generated by joining two frequent itemset of size $(k - 1)$. For example, at level 3, the pattern $\{a, b, c\}$ is generated by joining $\{a, b\}$ and $\{a, c\}$. After generating the candidate patterns, the support of the patterns is computed by scanning every transaction in the database and determining the frequent ones. In Fig. 2.4, a candidate patterns is shown in a box along with its support value. A frequent candidate is shown in a solid box, and an infrequent candidate is shown in a dotted box. An edge represents the join relationship between a candidate pattern of size k and a frequent pattern of size $(k-1)$ such that the latter is used to generate the earlier. The figure also illustrates the fact that a pair of frequent patterns are used to generate a candidate pattern, whereas no candidates are generated from an infrequent pattern.

2.1.1 Apriori Optimizations

Numerous optimizations were proposed for the *Apriori* algorithm [1] that are referred to as *AprioriTid* and *AprioriHybrid* respectively. In the *AprioriTid* algorithm, each transaction is replaced by a shorter transaction or null transaction) during the kth phase. Let the set of $k + 1$-candidates in $\mathcal{C}_{k+1}$ that are contained in transaction T be denoted by $\mathcal{R}(T, \mathcal{C}_{k+1})$. This set $\mathcal{R}(T, \mathcal{C}_{k+1})$ is added to a newly created transaction database $\mathcal{T}_k'$. If the set $\mathcal{R}(T, \mathcal{C}_{k+1})$ is null, then clearly, a number of different tradeoffs exist with the use of such an approach.

- Because each newly created transaction in $\mathcal{T}'_k$ is much shorter, this makes subsequent support counting more efficient.
- In some cases, no candidate may be a subset of the transaction. Such a transaction can be dropped from the database because it does not contribute to the counting of support values.
- In other cases, more than one candidate may be a subset of the transaction, which will actually increase the overhead of the algorithm. Clearly, this is not a desirable scenario.

Thus, the first two factors improve the efficiency of the new representation, whereas the last factor worsens it. Typically, the impact of the last factor is greater in the early iterations, whereas the impact of the first two factors is greater in the later iterations. Therefore, to maximize the overall efficiency, a natural approach would be to *not* use this optimization in the early iterations, and apply it only in the later iterations. This variation is referred to as the *AprioriHybrid* algorithm [1]. Another optimization proposed in [9] is that the support of many patterns can be inferred from those of key patterns in the data. This is used to significantly enhance the efficiency of the approach.

Numerous other techniques have been proposed that use different techniques to optimize the original implementation of the *Apriori* algorithm. As an example, the method in [1] and [44] share a number of similarities but are somewhat different at the implementation level. A work that combines the ideas from these different pieces of work is presented in [3].

2.2 DHP Algorithm

The DHP algorithm, also known as the *Direct Hashing and Pruning* method [50], was proposed soon after the *Apriori* method. It proposes two main optimizations to speed up the algorithm. The first optimization is to prune the candidate itemsets in each iteration, and the second optimization is to trim the transactions to make the support-counting process more efficient.

To prune the itemsets, the algorithm tracks partial information about candidate $(k+1)$-itemsets, while explicitly counting the support of candidate k-itemsets. During the counting of candidate k-itemsets, all $(k + 1)$ subsets of the transaction are found and hashed into a table that maintains the counts of the number of subsets hashed into each entry. During the phase of counting $(k + 1)$-itemsets, the counts in the hash table are retrieved for each itemset. Clearly, these counts are overestimates because of possible collisions in the hash table. Those itemsets for which the counts are below the user-specified support level are then pruned from consideration.

A second optimization proposed in DHP is that of transaction trimming. A key observation here is that if an item does not appear in at least k frequent itemsets in $\mathcal{F}_k$, then no frequent itemset in $\mathcal{F}_{k+1}$ will contain that item. This follows from the fact that there should be at least k (immediate) subsets of each frequent pattern in $\mathcal{F}_{k+1}$

containing a particular item that also occur in $\mathcal{F}_k$ and also contain that item. This implies that if an item does not appear in at least k frequent itemsets in $\mathcal{F}_k$, then that item is no longer relevant to further support counting for finding frequent patterns. Therefore, that item can be trimmed from the transaction. This reduces the width of the transaction, and increases the efficiency of processing. The overhead from the data structures is significant, and most of the advantages are obtained for patterns of smaller length such as 2-itemsets. It was pointed out in later work [46, 47, 60] that the use of triangular arrays for support counting of 2-itemsets in the context of the *Apriori* method is even more efficient than such an approach.

2.3 Special Tricks for 2-Itemset Counting

A number of special tricks can be used to improve the effectiveness of 2-itemset counting. The case of 2-itemset counting is special and is often similar for the case of join-based and tree-based algorithms. As mentioned above, one approach is to use a triangular array that maintains the counts of the k-patterns explicitly. For each transaction, a nested loop can be used to explore all pairs of items in the transaction and increment the corresponding counts in the triangular array. A number of caching tricks can be used [5] to improve data locality access during the counting process. However, if the number of possible items are very large, this will still be a very significant overhead because it is needed to maintain an entry for each pair of items. This is also very wasteful, if many of the 1-items are not frequent, or some of the 2-item counts are zero. Therefore, a possible approach would be to first prune out all the 1-items which are not frequent. It is simply not necessary to count the support of a 2-itemset unless both of its constituent items are frequent. A hash table can then be used to maintain the frequency counts of the corresponding 2-itemsets. As before, the transactions are explored in a double nested loops, and all pairs of items are hashed into the table, with the caveat, that each of the individual items must be frequent. The set of itemsets which satisfy the support requirements are reported.

2.4 Pruning by Support Lower Bounding

Most of the pruning tricks discussed earlier prune itemsets when they are guaranteed *not* meet the required support threshold. It is also possible to skip the counting process for an itemset if the itemset is guaranteed to meet the support threshold. Of course, the caveat here is that the exact support of that itemset will not be available, beyond the knowledge that it meets the minimum threshold. This is sufficient in the case of many applications.

Consider two k-itemsets A and B that have $k-1$ items $A \cap B$ in common. Then, the union of the items in A and B, denoted by $A \cup B$ will have exactly $k+1$ items. Then, if $sup(\cdot)$ represent the support of an itemset, then the support of $A \cup B$ can be lower bounded as follows:

$$sup(A \cup B) \geq sup(A) + sup(B) - sup(A \cap B) \tag{2.1}$$

This condition follows directly from set-theoretic considerations. Thus, the support of $(k+1)$-candidates can be lower bounded in terms of the (already computed) support values of itemsets of length k or less. If the computed value on the right-hand side is greater than the required minimum support, then the counting of the candidate does not need to be performed explicitly, and therefore considerable savings can be achieved. An example of a method which uses this kind of pruning is the *Apriori_LB* method [10].

Another interesting rule is that if the support of an itemset X is the same as that of $X \cup Y$, then for any superset $X' \supseteq X$, it is the case that the support of the itemset X' is the same as that of $X' \cup Y$. This rule can be shown directly as a corollary of the equation above. This is very useful in a variety of frequent pattern mining algorithms. For example, once the support of $X \cup \{i\}$ has been shown to be the same as that of X, then, for any superset X' of X, it is no longer necessary to explicitly compute the support of $X' \cup \{i\}$, after the support of X' has already been computed. Such optimizations have been shown to be quite effective in the context of many frequent pattern mining algorithms [13, 51, 17]. As discussed later, this trick is not exclusive to join-based algorithms, and is often used effectively in tree-based algorithms such as *MaxMiner*, and *MAFIA*.

2.5 Hypercube Decomposition

One feasible way to reduce the computation cost of support counting is to find support of multiple frequent patterns at one time. LCM [66] devise a technique referred to as hypercube decomposition in this purpose. The multiple itemsets obtained at one time, comprise a hypercube in the itemset lattice. Suppose that P is a frequent pattern, $tidset(P)$ contains the transactions that P is part of, and tail(P) denotes the latest item extension to the itemset P. $H(P)$ is the set of items e satisfying $e > tail(P)$ and $tidset(P) = tidset(P \cup e)$. The set $H(P)$ is referred to as the hypercube set. Then, for any $P' \subseteq H(P)$, $tidset(P \cup P') = tidset(P)$ is true, and $P \cup P'$ is frequent. The work in [66] uses this property in the candidate generation phase. For two itemsets P and $P \cup P'$, we say that P'' is between P and $P \cup P'$ if $P \subseteq P'' \subseteq P \cup P'$. In the phase with respect to P, we output all P'' between P and $P \cup H(P)$. This technique saves significant time in counting.

3 Tree-Based Algorithms

The tree-based algorithm is based on set-enumeration concepts. The candidates can be explored with the use of a subgraph of the lattice of itemsets (see Fig. 2.2), which is also referred to as the lexicographic tree or enumeration tree [5]. These terms will,

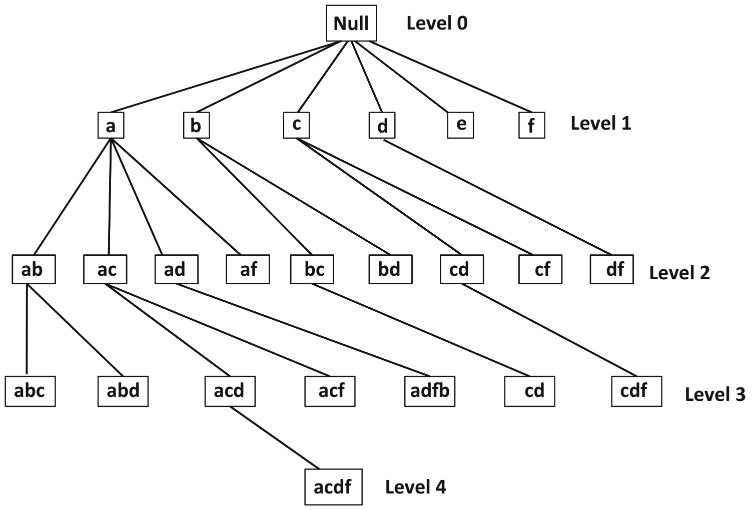

Fig. 2.5 The lexicographic tree (also known as enumeration tree)

therefore, be used interchangeably. Thus, the problem of frequent itemset generation is equivalent to that of constructing the enumeration tree. The tree can be grown in a wide variety of ways such as breadth-first or depth-first order. Because most of the discussion in this section will use this structure as a base for algorithmic development, this concept will be discussed in detail here. The main characteristic of tree-based algorithms is that the enumeration tree (or lexicographic tree) provides a certain order of exploration that can be extremely useful in many scenarios.

It is assumed that a lexicographic ordering exists among the items in the database. This lexicographic ordering is essential for efficient set enumeration without repetition. To indicate that an item i occurs lexicographically earlier than j, we will use the notation $i \leq_L j$. The lexicographic tree is an abstract representation of the large itemsets with respect to this ordering. The lexicographic tree is defined in the following way:

- A node exists in the tree corresponding to each large itemset. The root of the tree corresponds to the *null* itemset.
- Let $I = \{i_1, \ldots i_k\}$ be a large itemset, where $i_1, i_2 \ldots i_k$ are listed in lexicographic order. The parent of the node I is the itemset $\{i_1, \ldots i_{k-1}\}$.

This definition of ancestral relationship naturally defines a tree structure on the nodes that is rooted at the *null* node. A frequent 1-extension of an itemset such that the last item is the contributor to the extension will be called a *frequent lexicographic tree extension*, or simply a tree extension. Thus, each edge in the lexicographic tree corresponds to an item which is the frequent lexicographic tree extension to a node. The frequent lexicographic extensions of node P are denoted by $E(P)$. An example of the lexicographic tree is illustrated in Fig. 2.5. In this example, the frequent lexicographic extensions of node a are b, c, d, and f.

Let Q be the immediate ancestor of the itemset P in the lexicographic tree. The set of *prospective branches* of a node P is defined to be those items in $E(Q)$ which occur lexicographically after the node P. These are the *possible* frequent lexicographic extensions of P. We denote this set by $F(P)$. Thus, we have the following relationship: $E(P) \subseteq F(P) \subset E(Q)$. The value of $E(P)$ in Fig. 2.5, when $P = ab$ is $\{c, d\}$. The value of $F(P)$ for $P = ab$ is $\{c, d, f\}$, and for $P = af$, $F(P)$ is empty.

It is important to point out that virtually all non-maximal and maximal algorithms, starting from *Apriori*, can be considered enumeration-tree methods. In fact, there are few frequent pattern mining algorithms which do not use the enumeration tree, or a subset thereof (in maximal pattern mining) for frequent itemset generation. However, the *order of exploration* of the different algorithms of the lexicographic tree is quite different. For example, *Apriori* uses a breadth-first strategy, whereas other algorithms discussed later in this chapter use a depth-first strategy. Some methods are explicit about the relationship about the candidate generation process with the enumeration tree, whereas others, such as *Apriori*, are not. For example, by examining Fig. 2.4, it is evident that *Apriori* candidates can be generated by joining two frequent siblings of a lexicographic tree. In fact, all candidates can be generated in an exhaustive and non-redundant way by joining frequent siblings. For example, the two itemsets $acdfh$ and $acdfg$ are siblings, because they are children of the node $acdf$. By joining them, one obtains the candidate pattern $acdfgh$. Thus, while the *Apriori* algorithm is a join-based algorithm, it can also be explained in terms of the enumeration tree.

Parts of the enumeration tree may be removed by some of the algorithms by pruning methods. For example, the *Apriori* algorithm uses a levelwise pruning trick. For *maximal* pattern mining the advantages gained from pruning tricks can be very significant. Therefore, the number of candidates in the execution tree of different algorithms is different only because of pruning optimization tricks. However, some methods are able to achieve better *counting* strategies by using the structure of the enumeration tree to avoid re-doing the counting work already done for k-candidates to $(k+1)$-candidates. Therefore, explicitly introducing the enumeration tree is helpful because it allows a more flexible way to visualize candidate exploration strategies than join-based methods. The explicit introduction of the enumeration tree also helps in understanding whether the gains in different algorithms arise as a result of fewer number of candidates, or whether they arise as a result of better counting strategies.

3.1 AIS Algorithm

The original *AIS* algorithm [2] is a simple version of the lexicographic-tree algorithm, though it is not directly presented as such. In this approach, the tree is constructed in levelwise fashion and the corresponding itemsets at a given level are counted with the use of the transaction database. The algorithm does not use any specific optimizations to improve the efficiency of the counting process. As will be discussed later, a variety of methods can be used to further improve the efficiency of tree-based algorithms. Thus, this is a primitive approach that explores the entire search space with no optimization.

3.2 TreeProjection Algorithms

Two variants of an algorithm which use recursive projections of the transactions down the lexicographic tree structure are proposed in [5] and [4], respectively. The goal of using these recursive projections is to reuse the counting work down at a given level for lower levels of the tree. This reduces the counting work at the lower levels by orders of magnitude, as long as it is possible to successfully manage the memory requirements of the projected transactions. The main difference between the different versions of *TreeProjection* is the exploration strategy used. *TreeProjection* can be viewed as a generic framework that advocates the notion of database projection, in the context of several different strategies for constructing the enumeration tree, such as a breadth-first, depth-first, or a combination of the two. The depth-first version, described in detail in [4], also incorporates maximal pruning, though the disabling of the pruning options can also materialize all the patterns. The breadth-first and depth-first algorithms have different advantages. The former allows level-wise pruning which is not possible in depth-first methods though it is often not used in projection-based methods. The depth-first version allows better memory management. The depth-first approach works best when the itemsets are very long, and it is desirable to quickly discover maximal patterns, so that portions of the lexicographic tree can be pruned off quickly during exploration and it can also be used for discovering all patterns including non-maximal ones. When all patterns are required, including non-maximal ones, the primary difference between different strategies is not one of the size of the candidate space, but that of effective memory management of the projected transactions. This is because the size of the candidate space is defined by the size of the enumeration tree, which is fixed, and is agnostic to the strategy used for tree exploration. On the other hand, memory management of projected transactions is easier with the depth-first strategy because one only needs to maintain a small number of projected transaction sets along the depth of the tree. The notion of database projection is common to *TreeProjection* and *FP-growth*, and helps reduce the counting work by restricting the size of the database used for support counting. *TreeProjection* was developed independently from *FP-growth*. While the *FP-growth* paper provides a brief discussion of *TreeProjection*, this chapter will provide a more detailed discussion of the similarities and differences between the two methods. One major difference between the two methods is that the internal representation of the corresponding projected databases is different in the two cases.

The basic database projection approach is very similar in both cases of *TreeProjection* and *FP-growth*. An important observation is that if a transaction is not relevant for counting at a given node in the enumeration tree, then it will not be relevant for counting in any descendent of that node. Therefore, only those transactions are retained that contain all items in P for counting at the node P in the projected transactions. Note that this set strictly reduces as we move to lower levels of the tree, and the set of relevant transactions at the lower level of the enumeration tree is a subset of the set at a higher level. Furthermore, only the presence of items corresponding to the candidate extensions of a node are relevant for counting at any of the subtrees rooted

Fig. 2.6 Enumeration tree exploration

```
Algorithm ExplorePrefix(Database: $\mathcal{T}$, Minimum Support: $s$,
        Current Pattern Prefix: $P$)
begin
  Count support of 1-items in $\mathcal{T}$;
  Remove infrequent items from $\mathcal{T}$;
    for each frequent item $i$ in $\mathcal{T}$ do
    begin
      Append $i$ to end of $P$ and add to
          set of frequent patterns;
      Construct conditional database $\mathcal{T}_i$
       with all transactions in $\mathcal{T}$ containing item $i$;
      Remove items lexicographically $\leq i$ from $\mathcal{T}_i$;
      ExplorePrefix($\mathcal{T}_i$, $s$, $P \cup \{i\}$);
    end
end
```

at that node. Therefore, the database is also projected in terms of attributes, in which only items which are candidate extensions at a node are retained. The candidate set $F(P)$ of item extensions of node P is a very small subset of the universe of items at lower levels of the enumeration tree. In fact, even the items in the node P need not be retained explicitly in the transaction, because they are known to always be present in all the selected transactions based on the first condition. This projection process is performed recursively in top–down fashion down the enumeration tree for counting purposes, where lower level nodes inherit the projections from higher level nodes and add one additional item to the projection at each level. The idea of this inheritance-based approach is that the projected database remembers the counting work done at higher levels of the enumeration tree by (successively) removing irrelevant transactions and irrelevant items at each level of the projection. Such an approach works efficiently because it never repeats the counting work which has already been done at the higher levels. Thus, the primary savings in the strategy arise from avoiding repetitive and wasteful counting.

A bare-bones depth-first version of *TreeProjection*, that is similar to *DepthProject*, but without maximal pruning, is described in Fig. 2.6. A more detailed description with maximal pruning and other optimizations is provided later in this chapter. Because the algorithm is described recursively, the current prefix P (node of the lexicographic tree) being extended is one of the arguments to the algorithm. In the initial call, the value of P is *null* because one intends to determine all frequent descendants at the root of the lexicographic tree. This algorithm recursively extends frequent prefixes and maintains only the transaction database relevant to the prefix. The frequent prefixes are extended by determining the items i that are frequent in $\mathcal{T}$. Then the itemset $P \cup \{i\}$ is reported. The extension of the frequent prefix can be viewed as a recursive call at a node of the enumeration tree. Thus, at a given enumeration tree node, one now has a completely independent problem of extending the prefix with the projected database that is relevant to all descendants of that node. The conditional database $\mathcal{T}_i$ refers to the subset of the original transaction database $\mathcal{T}$ corresponding to transactions containing item i. Furthermore, the item i and any item occurring lexicographically earlier to it is not retained in the database because these items are not relevant to counting the extensions of $P \cup \{i\}$. This independent problem is similar in structure to the original problem, and can be solved recursively. Although it is natural to use recursion for the depth-first versions of *TreeProjection*, the breadth-first versions are not defined recursively. Nevertheless, the breadth-first versions explore a pattern space of the same size as the depth-first versions, and are no different either in terms of the tree size or the counting work done over the entire algorithm. The major challenge in the breadth-first version is in maintaining the projected transactions along the breadth of the tree, which is storage-intensive. It is shown in [5], how many of these issues can be resolved with the use of a combination of exploration strategies for tree growth and counting. Furthermore, it is also shown in [5] how breadth-first and depth-first methods may be combined.

Note that this concept of database projection is common between *TreeProjection* and *FP-growth* although there are some differences in the internal representation of the projected databases. The aforementioned description is designed for discovering all patterns, and does not incorporate maximal pattern pruning. When generating *all* the itemsets, the main advantage of the depth-first strategy over the breadth-first strategy is that it is less memory intensive. This is because one does not have to *simultaneously* handle the large number of candidates along the breadth of the enumeration tree at any point in the course of algorithm execution when combined with counting data structures. The overall size of the candidate space is fixed, and defined by the size of the enumeration tree. Therefore, over the entire execution of the algorithm, there is no difference between the two strategies in terms of search space size, beyond memory optimization.

Projection-based algorithms, such as *TreeProjection*, can be implemented either recursively or non-recursively. Depth-first variations of projection strategies, such as *DepthProject* and *FP-growth*, are generally implemented recursively in which a particular prefix (or suffix) of frequent items is grown recursively (see Fig. 2.6). For recursive variations, the structure and size of the recursion tree is the same as the enumeration tree. Non-recursive variations of *TreeProjection* methods directly present the projection-based algorithms in terms of the enumeration tree by storing projected transactions at the nodes in the enumeration tree. Describing projection strategies directly in terms of the enumeration tree is helpful, because one can use the enumeration tree explicitly to optimize the projection. For example, one does not need to project at every node of the enumeration tree, but project only when the size of the database reduces by a particular factor with respect to the nearest ancestor node where the last projection was stored. Such optimizations can reduce the space-overhead of repeated elements in the projected databases at different levels of the enumeration (recursion) tree. It has been shown how to use this optimization in different variations of *TreeProjection*. Furthermore, breadth-first variations of the strategy are naturally defined non-recursively in terms of the enumeration tree. The recursive depth-first versions may be viewed either as divide-and-conquer strategies (because they recursively solve a set of smaller subproblems), or as projection-based counting reuse strategies. The notion of projection-based counting reuse clearly describes how computational savings are achieved in both versions of the algorithm.

When generating *maximal* patterns, the depth-first strategy has clear advantages in terms of pruning as well. We refer the reader to a detailed description of the *DepthProject* algorithm, described later in this chapter. This description describes how several specialized pruning techniques are enabled by the depth-first strategy for maximal pattern mining. The *TreeProjection* algorithm has also been generalized to sequential pattern mining [31]. There are many different types of data structures that may be used in projection-style algorithms. The choice of data structure is sensitive to the data set. Two common choices that are used with *TreeProjection* family of algorithms are as follows:

1. *Arrays:* In this case, the projected database is maintained as 2-dimensional array. One of the dimensions of the array is equal to the number of relevant transactions and the other dimension is equal to the number of relevant items in the projected database. Both dimensions of the projected database reduce from top level to lower levels of the enumeration tree with successive projection.
2. *BitStrings:* In this case, the projected database is maintained as a 0–1 bit string whose width is fixed to the total number of frequent 1-items, but the number of projected transactions reduces with successive projection. Such an approach loses the power of item-wise projection, but this is balanced by the fact that the bit-strings can be used more efficiently for counting operations.
   Assume that each transaction T contains n bits, and can therefore be expressed in the form of $\lceil n/8 \rceil$ bytes. Each byte of the transaction contains the information about the presence or absence of eight items, and the integer value of the corresponding bitstring can take on any value from 0 to $2^8 - 1 = 255$. Correspondingly, for each byte of the (projected) transaction at a node, 256 counters are maintained and a value of 1 is added to the counter corresponding to the integer value of that transaction byte. This process is repeated for each transaction in the projected database at node P. Therefore, at the end of this process, one has $256 * \lceil d/8 \rceil$ counts for the d different items. At this point, a postprocessing phase is initiated in which the support of an item is determined by adding the counts of the $256/2 = 128$ counters which take on the value of 1 for that bit. Thus, the second phase requires $128 * d$ operations only, and is independent of database size. The first phase, (which is the bottleneck) is the improvement over the naive counting method because it performs only one operation for each *byte* in the transaction, which contains eight items. Thus, the method would be a factor of eight faster than the naive counting technique, which would need to scan the entire bitstring. Projection is also very efficient in the bitstring representation with simple AND operations.

The major problem with fixed width bitstrings is that they are not efficient representations at lower levels of the enumeration tree at which only a small number of items are relevant, and therefore most entries in these bitstrings are 0. One approach to speed this up is to perform the item-wise projection only at selected nodes in the tree, when the reduction in the number of items from the last ancestor at which the item-wise projection was performed is at particular multiplicative factor. At this point, a shorter bit string is used for representation for the descendants at that node,

Table 2.2 Vertical representation of transactions. Note that the support of itemset *ab* can be computed as the length of the intersection of the *tidlists* of *a* and *b*

Item	tidlist
a	1, 2, 3, 5
b	1, 2, 4, 5
c	1, 2, 5
d	1, 2, 5
e	1, 4, 5
f	2, 3, 4
g	3, 4
h	2, 5

until the width of the bitstring is reduced even further by the same multiplicative factor. This ensures that the bit strings representations are not sparse and wasteful.

The key issue here is that different representations provide different tradeoffs in terms of memory management and efficiency. Later in this chapter, an approach called *FP-growth* will be discussed which uses the trie data structure to achieve compression of projected transactions for better memory management.

3.3 Vertical Mining Algorithms

The vertical pattern mining algorithms use a vertical representation of the transaction database to enable more efficient counting. The basic idea of the vertical representation is that one can express the transaction database as an inverted list. In other words, for each transaction identifiers, one can have a list of items that are contained in it. This is referred to as a *tidset* or *tidlist*. An example of a vertical representation of the transactions in Table 2.1 is illustrated in Table 2.2.

The key idea in vertical pattern mining algorithms is that the support of k-patterns can be computed by intersection of the underlying *tidlists*. There are two different ways in which this can be done.

- The support of a k-itemset can be computed as a k-way set intersection of the lists of the individual items.
- The support of a k-itemset can be computed as an intersection of the *tidlists* two $(k-1)$-itemsets that join to that k-itemset.

The latter approach is more efficient. The credit for both the notion of vertical tidlists and the advantages of recursive intersection of tidlists is shared by the *Monet* [56] and the *Partition* algorithms [57]. Not all vertical pattern mining algorithms use an enumeration tree concept to describe the algorithm. Many of the algorithms directly use joins to generate a $(k+1)$-candidate pattern from a frequent k-pattern, though even a join-based algorithm, such as *Apriori*, can be explained in terms of an enumeration tree. Many of the later variations of vertical methods use an enumeration tree concept to explore the lattice of itemsets more carefully and realize the full power of the vertical approach. The indvidual ensemble component of Savasere et al.'s [57] *Partition* algorithm is the progenitor of all vertical pattern mining algorithms today, and the original *Eclat* algorithm is a memory-optimized and candidate partitioned version of this Apriori-like algorithm.

3.3.1 Eclat

Eclat uses a breadth-first approach like Savasere et al.'s algorithm [57] on lattice partitions, after partitioning the candidate set into disjoint groups, using a candidate partitioning approach similar to earlier parallel versions of the *Apriori* algorithm. The *Eclat* [71] algorithm is best described with the concept of an enumeration tree because of the wide variation in the different strategies used by the algorithm. An important contribution of *Eclat* [71] is to recognize the earlier pioneering work of the *Monet* and *Partition* algorithms [56, 57] on recursive intersection of tid lists, and propose many efficient variants of this paradigm.

Different variations of *Eclat* explore the candidates in different strategies. The earliest description of *Eclat* may be found in [74]. A journal paper exploring different aspects of *Eclat* may be found in [71]. In the earliest versions of the work [74], a breadth-first strategy is used. The journal version in [71] also presents experimental results for only the breadth-first strategy, although the possibility of a depth-first strategy is mentioned in the paper. Therefore, the original *Eclat* algorithm should be considered a breadth-first algorithm. More recent depth-first versions of *Eclat*, such as *dEclat*, use recursive *tidlist* intersection with differencing [72], and realize the full benefit of the depth-first approach. The *Eclat* algorithm, as presented in [74], uses a levelwise strategy in which all $(k+1)$-candidates within a lattice partition are generated from frequent k-patterns in level-wise fashion, as in *Apriori*. The *tidlists* are used to perform support counting. The frequent patterns are determined from these *tidlists*. At this point, a new levelwise phase is initiated for frequent patterns of size $(k+1)$.

Other variations and depth-first exploration strategies of *Eclat*, along with experimental results, are presented in later work such as *dEclat* [72]. The *dEclat* work in [72] presents some additional enhancements such as *diffsets* to improve counting. In this chapter, we present a simplified pseudo-code of this version of *Eclat*. The algorithm is presented in Fig. 2.8. The algorithm is structured as a recursive algorithm. A pattern set $\mathcal{FP}$ is part of the input, and is set to the set of all frequent 1-items at the top level call. Therefore, it may be assumed that, at the top level, the set of frequent 1-items and $tidlists$ have already been computed, though this computation is not shown in the pseudocode. In each recursive call of *Eclat*, a new set of candidates $\mathcal{FP}_i$ is generated for every pattern (itemset) P_i, which extends the itemset by one unit. The support of a candidate is determined with the use of *tidlist* intersection. Finally, if P_i is frequent, it is added to a pattern set $\mathcal{FP}_i$ for the next level.

Figure 2.7 illustrates the itemset generation tree with support computation by *tidlist* intersection for the sample database from Table 2.1. The corresponding *tidlists* in the tree are also illustrated. All infrequent itemsets in each level are denoted by dotted, and bordered rectangles. For example, an itemset ab is generated by joining b to a. The *tidlist* of (a) is $\{1, 2, 3, 5\}$, and the *tidlist* of b is $\{1, 2, 4, 5\}$. We can determine the support of ab by intersecting the two *tidlists* to obtain the *tidlist* $\{1, 2, 5\}$ of these candidates. Therefore, the support of ab is given by the length of this *tidlist*, which is 3.

Further gains may be obtained with the use of the notion of *diffsets* [72]. This approach realizes the true power of vertical pattern mining. The basic idea, in *diffsets* is to maintain only the portion of the *tidlists* at a node, that correspond to the change in the inverted list from the parent node. Thus, the *tidlists* at a node can be reconstructed by examining the *tidlists* at the ancestors of a node in the tree. The major advantage

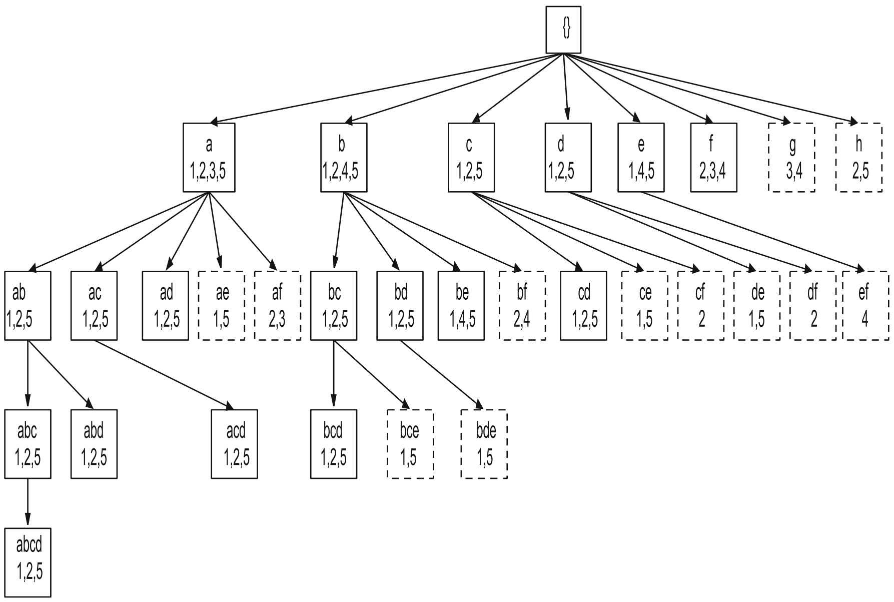

Fig. 2.7 Execution of *Eclat*

Fig. 2.8 The *Eclat* algorithm

```
Algorithm $Eclat(\mathcal{FP}$, Support: $s)$
begin
   for each $P_i \in \mathcal{FP}$ do
   begin
      $\mathcal{FP}_i = \{\}$
      for each $P_j \in \mathcal{FP}$, such that $j > i$ do
      begin
         $P_{ij} = P_i \cup P_j$
         $\text{tidset}(P_{ij}) = \text{tidset}(P_i) \cap \text{tidset}(P_j)$
         $\text{support}(P_{ij}) = |\text{tidset}(P_{ij})|$
         if $(\text{support}(P_{ij}) \geq s)$
            Add $P_{ij}$ to $\mathcal{FP}_i$;
      end
      $Eclat(\mathcal{FP}_i, s)$
   end
end
```

of *diffsets* is that they save significant storage in requirements in terms of the size of the data structure required (Fig. 2.8).

Fig. 2.9 Suffix-based pattern exploration

```
Algorithm SuffixGrowth(Database of Frequent Items: $\mathcal{T}$,
        Minimum Support: $s$, Current Pattern Suffix: $P$)
begin
  for each item $i$ in $\mathcal{T}$ do
  begin
    Append $i$ to beginning of $P$ and add to
        set of frequent patterns;
    Let $\mathcal{T}_i$ be transactions of $\mathcal{T}$ containing $i$;
    Remove any item lexicographically $\geq i$ from $\mathcal{T}_i$;
    Remove infrequent items in $\mathcal{T}_i$;
    if $\mathcal{T}_i \neq \phi$ SuffixGrowth($\mathcal{T}_i$, $s$, $\{i\} \cup P$);
  end
end
```

3.3.2 VIPER

The *VIPER* algorithm [58] uses a vertical approach to mining frequent patterns. The basic idea in the *VIPER* algorithm is ro represent the vertical database in the form of compressed bit vectors that are also referred to as *snakes*. These snakes are then used for efficient counting of the frequent patterns. The different compressed representation of the *tidlists* provide a number of optimization advantages that are leveraged by the algorithm. Intrinsically, *VIPER* is not very different from *Eclat* in terms of the basic counting approach. The major difference is in terms of the choice of the compressed bit vector representation, and the efficient handling of this representation. Details may be found in [58].

4 Recursive Suffix-Based Growth

In these algorithms recursive suffix-based exploration of the patterns is performed. Note that in most frequent pattern mining algorithms, the enumeration tree (execution tree) of patterns explores the patterns in the form of a lexicographic tree of itemsets built on the prefixes. Suffix-based methods use a different convention in which the suffixes of frequent patterns are extended. As in all projection-based methods, one only needs to use the transaction database containing itemset P in order to count itemsets that have the suffix P. Itemsets are extended from the suffix backwards. In each iteration, the conditional transaction database (or projected database) of transactions containing the current suffix P being explored is an input to the algorithm. Furthermore, it is assumed that the conditional database contains only frequent extensions of P. For the top-level call, the value of P is null, and the frequent items are determined using a single preprocessing pass that is not shown in the pseudo-code. Because each item is already known to be frequent, the frequent patterns $\{i\} \cup P$ can be immediately generated for each item $i \in \mathcal{T}$. The database is projected further to include only transactions containing i, and a recursive call is initiated with the pattern $\{i\} \cup P$. The projected database $\mathcal{T}_i$ corresponding to transactions containing $\{i\} \cup P$ is determined. Infrequent items are removed from $\mathcal{T}_i$. Thus, the transactions are recursively projected to reflect the addition of an item in the suffix. Thus, this is a smaller subproblem that can be solved recursively. The *FP-growth* approach uses the suffix-based pattern exploration, as illustrated in Fig. 2.9. In addition, the *FP-growth* approach uses an efficient data structure, known as the FP-Tree to represent the conditional transaction database $\mathcal{T}_i$ with the use of compressed *prefixes*. The *FP-Tree* will be discussed in more detail in a later section. The suffix in the top level call to the algorithm is the null itemset.

Recursive suffix-based exploration of the pattern space is, in principle, no different from prefix-based exploration of the enumeration tree space with the ordering of the items reversed. In other words, by using a reverse ordering of items, suffix-based recursive pattern space exploration can be simulated with prefix-based enumeration tree exploration. Indeed, as discussed in the last section, prefix-based enumeration tree methods order items from the least frequent to the most frequent, whereas the suffix-based methods of this section order items from the most frequent to the least frequent, to account for this difference. Thus, suffix-based recursive growth has an execution tree that is identical in structure to a prefix-based enumeration tree. This is a difference only of convention, but it does not affect the pattern space that is explored.

It is instructive to compare the suffix-based exploration with the pseudocode of the prefix-based *TreeProjection* algorithm in Fig. 2.6. The two pseudocodes are structured differently because the initial pre-processing pass of removing frequent items is not assumed in the *TreeProjection* algorithm. Therefore, in each recursive call of the prefix-based *TreeProjection*, frequent itemsets must be counted before they are reported. In suffix-based exploration, this step is done as a preprocessing step (for the top-level call) and just before the recursive call for deeper calls. Therefore, each recursive call always starts with a database of frequent items. This is, of course, a difference in terms of how the recursive calls are structured but is not different in terms of the basic search strategy, or the amount of overall computational work required, because infrequent items need to be removed in either case. A few other key differences are evident:

- *TreeProjection* uses database projections on top of a prefix-based enumeration tree. Suffix-based recursive methods have a recursion tree whose structure is similar to an enumeration tree on the frequent suffixes instead of the prefixes. The removal of infrequent items from $\mathcal{T}_i$ in *FP-growth* is similar to determining which branches of the enumeration tree to extend further.
- The use of suffix-based exploration is a difference only of convention from prefix-based exploration. For example, after reversing the item order, one might implement *FP-growth* by growing patterns on the prefixes, but constructing a compressed FP-Tree on[1] the suffixes. The resulting exploration order and execution in the two different implementations of *FP-growth* will be identical, but the latter can be more easily related to traditional enumeration tree methods.

[1] The resulting FP-Tree will be a suffix-based trie.

- Various database projection methods are different in terms of the specific data structures used for the projected database. The different variations of *TreeProjection* use arrays and bit strings to represent the projected database. The *FP-growth* method uses an FP-Tree. The FP-Tree will be discussed in the next section. Later variations of FP-Tree also use combinations of arrays and pointers to represent the projected database. Some variations, such as *OpportuneProject* [38], combine different data structures in an optimized way to obtain the best result.
- Suffix-based recursive growth is inherently defined as a depth-first strategy. On the other hand, as is evident from the discussion in [5], the specific choice of exploration strategy on the enumeration tree is orthogonal to the process of database projection. The overall size of the enumeration tree is the same, no matter how it is explored, unless maximal pattern pruning is used. Thus, *TreeProjection* explores a variety of strategies such as breadth-first and depth-first strategies, with no difference to the (overall) work required for counting. The major challenge with the breadth-first strategy is the simultaneous maintenance of projected transaction sets along the breadth of the tree. The issue of effective memory management of breadth-first strategies is discussed in [5], which shows how certain optimizations such as cache-blocking can improve the effectiveness in this case. Breadth-first strategies also allow certain kinds of pruning such as level-wise pruning.
- The major advantages of depth-first strategies arise in the context of maximal pattern mining. This is because a depth-first strategy discovers the maximal patterns very early, which can be used to prune the smaller non-maximal patterns. In this case, the size of the search space explored truly reduces because of a depth-first strategy. This issue is discussed in the section on maximal pattern mining. The advantages for maximal pattern mining were first proposed in the context of the *DepthProject* algorithm [4].

Next, we will describe the FP-Tree data structure that uses compressed representations of the transaction database for more efficient counting.

4.1 The FP-Growth Approach

The *FP-growth* approach combines suffix-based pattern exploration with a compressed representation of the projected database for more efficient counting. The prefix-based FP-Tree is a compressed representation of the database which is built by considering a fixed order among the items in an itemset [32]. This tree is used to represent the conditional transaction sets $\mathcal{T}$ and $\mathcal{T}_i$ of Fig. 2.9. An FP-Tree may be viewed as a prefix-based trie data structure of the transaction database of frequent items. Just as each node in a trie is labeled with a symbol, a node in the FP-Tree is labeled with an item. In addition, the node holds the support of the itemset defined by the items of the nodes that are on the path from the root to u. By consolidating the prefixes, one obtains compression. This is useful for effective memory management. On the other hand, the maintenance of counts and pointers with the prefixes is an

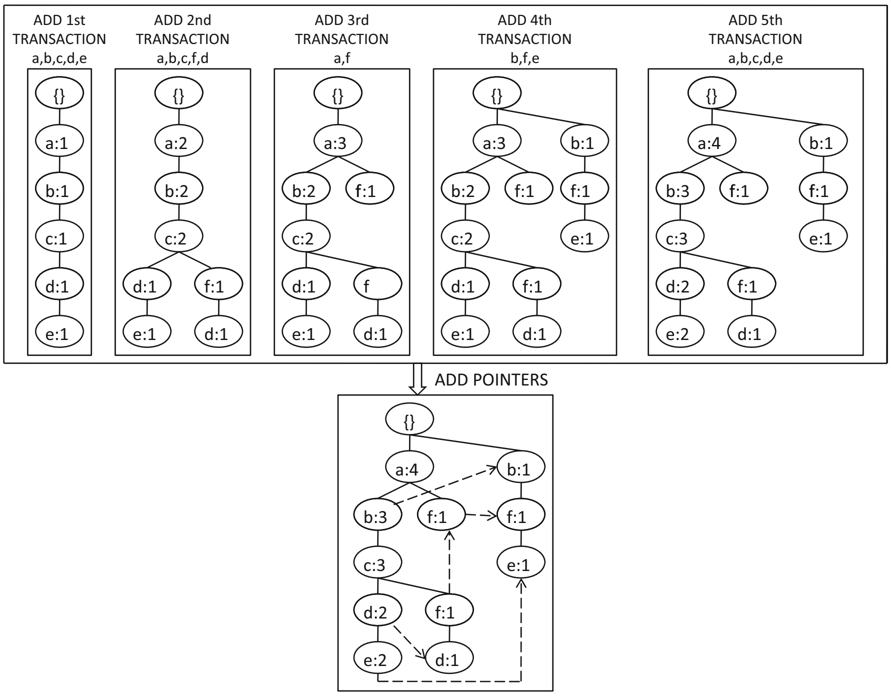
ADD 1st TRANSACTION a,b,c,d,e
ADD 2nd TRANSACTION a,b,c,f,d
ADD 3rd TRANSACTION a,f
ADD 4th TRANSACTION b,f,e
ADD 5th TRANSACTION a,b,c,d,e
ADD POINTERS
{}
a:1
b:1
c:1
d:1
e:1
a:2
b:2
c:2
f:1
a:3
a:4
b:3
c:3
d:2
e:2

Fig. 2.10 FP-Tree construction

additional overhead. This results in a different set of trade-offs as compared to the array representation.

The initial FP-Tree is constructed as follows. We start with the empty FP-Tree FPT. Before constructing the FP-Tree, the database is scanned and infrequent items are removed. The frequent items are sorted in decreasing order of support. The initial construction of FP-Tree is straightforward, and similar to how one might insert a string in a trie. For every insertion, the counts of the relevant nodes that are affected by the insertion are incremented by 1. If there has been any sharing of prefix between the current transaction t being inserted, and a previously inserted transaction then t will be in the same path until the common prefix. Beyond this common prefix, new nodes are inserted in the tree for the remaining items in t, with support count initialized to 1. The above procedure ends when all transactions have been inserted.

To store the items in the final FP-Tree, a list structure called header table is maintained. A chain of pointers threads through the occurrence of the item in the FP-Tree. Thus, this chain of pointers need to be constructed in addition to the trie data structure. Each entry in this table stores the item label and pointers to the node representing the leftmost occurrence of the item in the FP-Tree (first item in the pointer chain). The reason for maintaining these pointers is that it is possible to determine the conditional FP-Tree for an item by chasing the pointers for that item. An example of the initial construction of the FP-Tree data structure from a

Fig. 2.11 The *FP-growth* algorithm

```
Algorithm FP-growth(FP-Tree on Frequent Items: $FPT$,
      Minimum Support; $s$, Current Itemset Suffix: $P$)
begin
  if $FPT$ is a single path or empty
    for each combination $C$ of nodes in path do
      report all patterns $C \cup P$;
  else
  for each item $i$ in $FPT$ do
  begin
    Generate pattern $P_i = \{i\} \cup P$;
    report pattern $P_i$ as frequent;
    Use pointer-chasing to extract conditional
          prefix paths for item $i$;
    Construct conditional FP-Tree $FPT_i$ from conditional
          prefix paths after removing infrequent items;
    if ($FPT_i \neq \phi$) FP-growth($FPT_i$, $P_i$, $s$)
  end
end
```

database of five transactions is illustrated in Fig. 2.10. The ordering of the items is a, b, c, d, e, f. It is clear that a trie data structure is created, and the node counts are updated by the insertion of each transaction in the FP-Tree. Figure 2.10 also shows all the pointers between the different items. The sum of the counts on the items on this pointer path is the support of the item. This support is always larger than the minimum support because a full constructed FP-Tree (with pointers) contains only frequent items. The actual counting of the support of item-extensions and the removal of infrequent items must be done during conditional transaction database (and the relevant FP-Tree) creation. The pointer paths are not available during the FP-Tree creation process. For example, the item e has two nodes on this pointer path, corresponding to $e : 2$ and $e : 1$. By summing up these counts, a total count of three for the item e is obtained. It is not difficult to verify that three transactions contain the item e.

With this new compressed representation of the conditional transaction database of frequent items, one can directly extract the frequent patterns. The pseudo-code of the *FP-growth* algorithm is presented in Fig. 2.11. Although this pseudo-code looks much more complex to understand than the earlier pseudocode of Fig. 2.9, the main difference is that more details of the data structure (FP-Tree), used to represent the conditional transaction sets, have been added.

The algorithm accepts a FP-Tree FPT, current itemset suffix P and user defined minimum support s as input. The additional suffix P has been added to the parameter set P to facilitate the recursive description. At the top level call made by the user, the value of P is ϕ. Furthermore, the conditional FP-Tree is constructed on a database of frequent items rather than all the items. This property is maintained across different recursive calls.

For an FP-Tree FPT, the conditional FP-Trees are built for each item i in FPT (which is already known to be frequent). The conditional FP-Trees are constructed by chasing pointers for each item in the FP-Tree. This yields all the conditional prefix

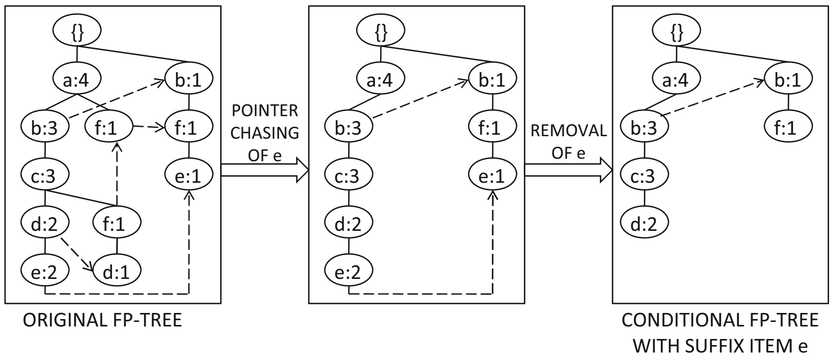

Fig. 2.12 Generating a conditional FP-Tree by pointer chasing

paths for the item i. The infrequent nodes from these paths are removed, and they are put together to create a conditional FP-Tree FPT_i. Because the infrequent items have already been removed from FPT_i the new conditional FP-Tree also contains only frequent items. Therefore, in the next level recursive call, any item from FPT_i can be appended to P_i to generate another pattern. The supports of those patterns can also be reconstructed via pointer chasing during the process of reporting the patterns. Thus, the current pattern suffix P is extended with the frequent item i appended to the front of P. This extended suffix is denoted by P_i. The pattern P_i also needs to be reported as frequent. The resulting conditional FP-Tree FPT_i is the compressed database representation of $\mathcal{T}_i$ of Fig. 2.9 in the previous section. Thus, FPT_i is a smaller conditional tree that contains information relevant only to the extraction of various prefix paths relevant to different items that will extend the suffix P_i further in the backwards direction. Note that infrequent items are removed from FPT_i during this step, which requires the support counting of all items in FPT_i. Because the pointers have not yet been constructed for FPT_i, the support of each item-extension of $\{i\} \cup P$ corresponding to the items in FPT_i must be explicitly determined by locating each instance of an item in FPT_i. This is the primary computational bottleneck step. The removal of infrequent items from FPT_i may result in a different structure of the FP-Tree in the next step.

Finally, if the conditional FP-Tree FPT_i is not empty, the *FP-growth* method is called recursively with parameters corresponding to the conditional FP-Tree FPT_i, extended suffix P_i, and minimum support s. Note that successive projected transaction databases (and corresponding conditional FP-Trees) in the recursion will be smaller because of the recursive projection. The base case of the recursion occurs when the entire FP-Tree is a single path. This is likely to occur when the projected transaction database becomes small enough. In that case, *FP-growth* determines all combinations of nodes on this path, appends the suffix P to them, and reports them.

An example of how the conditional FP-Tree is created for a minimum support of 1 unit, is illustrated in Fig. 2.12. Note that if the minimum support were 2, then the right branch (nodes b and f) would not be included in the conditional FP-Tree. In this case, the pointers for item e are chased in the FP-Tree to create the conditional prefix paths of the relevant conditional transaction database. This represents all transactions containing *e*. The counts on the prefix paths are re-adjusted because many branches are pruned. The removal of infrequent items and that of the item *e* might lead to a conditional FP-Tree that looks very different from the conditional prefix-paths. These kinds of conditional FP-trees need to be generated for each conditional frequent item, although only a single item has been shown for the sake of simplicity. Note that, in general, the pointers may need to be recreated every time a conditional FP-Tree is created.

4.2 Variations

As the database grows larger, the construction of the FP-Tree become challenging both from runtime and space complexity. There have been many works [8, 24, 27, 29, 30, 36, 39, 55, 59, 61, 62] to tackle these challenges. These variations of *FP-growth* method can be classified into two categories. Methods belonging to the first category design memory-based mining process using a memory-resident data structure that holds partitioned database. Methods belonging to the second category improve the efficiency of the FP-Tree representation. In this subsection, we will present these approaches briefly.

4.2.1 Memory-Resident Variations

In the following, a number of different memory-resident variations of the basic *FP-growth* idea will be described.

CT-PRO Algorithm In this work [62], the authors introduced a new FP-Tree like data structure called Compact FP-Tree (CFP-Tree) that holds the same information as FP-Tree but with 50 % less storage. They also designed a mining algorithm called CT-PRO which follows a non-recursive procedure unlike *FP-growth*. As discussed earlier, during the mining process, *FP-growth* constructs many conditional FP-Trees, which becomes an overhead as the patterns get longer or the support gets lower. To overcome this problem, the *CT-PRO* algorithm divides the database into several disjoint projections where each projection is represented as a CFP-Tree. Then a non-recursive mining process is executed over each projection independently. Significant modifications were made to the header Table 4.1 data structure. In the original FP-Tree, the nodes store the support and item label. However, in the CFP-Tree, item labels are mapped to an increasing sequence of integers that is actually the index of the header table. The header table of CFP-Tree stores the support of each item. To compress the original FP-Tree, all identical subtrees are removed by accumulating them and storing the relevant information in the leftmost branch. The header table contains a pointer to each node on the leftmost branch of the CFP-Tree, as these nodes are roots of subtrees starting with different items.

The mining process starts from the pointers of the least frequent items in the header table. This prunes a large number of nodes at an early stage and shrinks the tree structure. By following the pointers to the same item, a projection of all transactions ending with the corresponding item is built. This projection is also represented as a CFP-Tree called local CFP-Tree. The local CFP-Tree is then traversed to extract the frequent patterns in the projection.

H-Mine Algorithm The authors in [54] proposed an efficient algorithm called *H-Mine*. It uses a memory efficient hyper-structure called H-Struct. The fundamental strategy of *H-Mine* is to partition the database and mine each partition in the memory. Finally, the results from different partitions are consolidated into global frequent patterns. An intelligent module of *H-Mine* is that it can identify whether the database is dense or sparse, and it is able to make dynamic choices between different data structures based on this identification. More details may be found in Chap. 3 on pattern-growth methods.

4.2.2 Improved Data Structure Variations

In this section, several variations of the basic algorithm by improving the underlying data structure will be described.

Using Arrays A significant part of the mining time in *FP-growth* is spent on traversing the tree. To reduce this time, the authors in [29] designed an array based implementation of *FP-growth*, named *FP-growth** which drastically reduces the traversal time of the mining algorithm. It uses the FP-Tree data structure in combination with an array-like data structure and it incorporates various optimization schemes. It should be pointed out that the *TreeProjection* family of algorithms also uses arrays, though the optimizations used are quite different.

When the input database is sparse, the array based technique performs well because the array saves the traversal time for all the items; moreover the initialization of the next level of FP-Trees is easier using an array. But in case of dense database, the tree base representation is more compact. To deal with the situation, *FP-growth** devises a mechanism to identify whether the database is sparse or not. To do so, *FP-growth** counts the number of nodes in each level of the tree. Based on experiments, they found that if the upper quarter of the tree contains less than 15% of the total number of nodes, then the database is most likely dense. Otherwise, it is sparse. If the database turns out to be sparse, *FP-growth** allocates an array for each FP-Tree in the next level of mining.

The nonordfp Approach This work [55] presented an improved implementation of the well known *FP-growth* algorithm using an efficient FP-Tree like data structure that allows faster allocation, traversal and optional projection. The tree nodes do not store their labels (item identifiers). There is no concept of header table. The data structure stores less administrative information in the tree node which allow the recursive step of mining without rebuilding the tree.

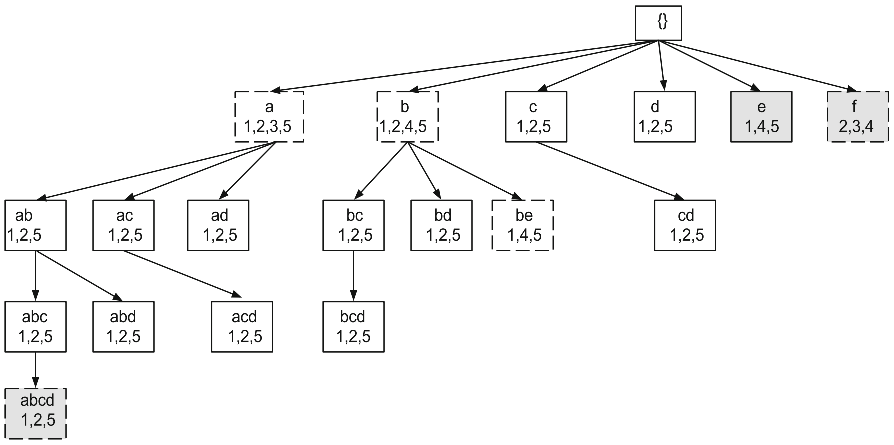

Fig. 2.13 Frequent, maximal and closed itemsets

5 Maximal and Closed Frequent Itemsets

One of the major challenges of frequent itemset mining is that, most of the itemsets mined are subset of the set of single length frequent items. Therefore, a significant amount of time is spent on counting redundant itemsets. One solution to this problem is to discover condensed representations of the frequent itemsets. It will be such representations that synopsizes the property of the set of itemsets completely or partially. The compact representation not only save computational and memory resource but also paved a much easier way towards knowledge discovery stage after mining. Another interesting observation by [53] was that, instead of mining the complete set of frequent itemsets and their associations, association mining only needs to find frequent closed itemsets and their corresponding rules. So, mining frequent closed itemset can fulfill the objectives of mining all frequent itemsets but with less redundancy and better efficiency and effectiveness in mining. In this section, we will discuss two types of condensed representation of itemset: maximal and closed frequent itemset.

5.1 Definitions

Maximal Frequent Itemset Suppose, $\mathcal{T}$ is the transaction database, $\mathcal{I}$ is the set of all items in the database and $\mathcal{F}$ is the set of all frequent itemsets. A frequent itemset $P \in \mathcal{F}$ is called maximal if it has no frequent superset. let $\mathcal{M}$ be the set of all frequent maximal itemsets, which is denoted by

$$\mathcal{M} = \{P \mid P \in \mathcal{F} \, and \, \nexists \, Q \supset P, \, such \, that \, Q \in \mathcal{F}\}$$

For the toy transaction database in Table 2.1 the frequent maximal itemsets at minimum support 3 are $abcd$, e, f, as illustrated in Fig. 2.13. All the rectangles filled with grey color represent maximal frequent patterns. As we can see in Fig. 2.4, that there are no frequent supersets of $abcd$, e or f.

Closed Frequent Itemset The closure operator γ induces an equivalence relation on the power set of items partitioning it into disjoint subsets called equivalence classes. The largest element with respect to the number of items in each equivalence class is called a closed itemset. A frequent itemset P is closed if $\gamma(P) = P$. From the closure property it can be said that both $\gamma(P)$ and P have the same tidset. In simpler terms, an itemset is closed if it does not have any frequent superset with the same support. A closed itemset $\mathcal{C}$ can be written as:

$$\mathcal{C} = \{P \mid P \in \mathcal{F} \, and \nexists Q \supset P, \, such \, that \, support(Q) = support(P)\}$$

Because maximal itemsets have no frequent superset, they are vacuously closed frequent itemsets. Thus, all maximal patterns are closed. However, there is a key difference between mining maximal itemsets and closed itemsets. Mining maximal itemsets loses information about the support of the underlying itemsets. On the other hand, mining closed itemsets does not lose any information about the support. The support of the missing subsets can be derived from the closed frequent pattern database. One way of viewing closed frequent patterns is as the maximal patterns from each *equi-support* group of frequent patterns. Closed frequent itemsets are a condensed representation of frequent itemsets that is lossless.

For the toy transaction database of Table 2.1 the frequent closed patterns are a, b, $abcd$, be for minimum support value of 3, as illustrated in Fig. 2.13. All the rectangles with dotted border represent closed frequent patterns. The remaining nodes in the tree (not filled and dotted border) represent frequent itemsets.

5.2 *Frequent Maximal Itemset Mining Algorithms*

In this subsection, we will discuss some of maximal frequent itemset mining algorithms.

5.2.1 MaxMiner Algorithm

The *MaxMiner* algorithm was the first algorithm that used a variety of optimizations to improve the effectiveness of tree explorations [10]. This algorithm is generally focussed on determining maximal patterns rather than all patterns. The author of [10] observed that it is usually sufficient to only report maximal patterns, when frequent patterns are long. This is because of the combinatorial explosion in examining all subsets of patterns. Although the exploration of the tree is still done in breadth-first fashion, a number of optimizations are used to improve the efficiency of exploration:

- The concept of *lookaheads* is defined. Let $F(P)$ be the set of candidate items that might extend node P. Before counting, it is checked whether $F \cup F(P)$ is a subset of any of the frequent patterns found so far. If such is indeed the case, then it is known that the entire subtree rooted at P is frequent, and can be pruned from consideration (for maximal pattern mining). During counting the support of individual item extensions of P, the support of $P \cup F(P)$ is also determined. If the set $P \cup F(P)$ is frequent, then it is known that all itemsets in the entire subset rooted at that node are frequent. Therefore, the tree does not need to be explored further, and can be pruned.
- The support lower bounding trick discussed earlier can be used to quickly determine patterns which are frequent without explicit counting. The counts of extensions of nodes can be determined without counting in many cases, where the count does not change by extending an item.

It has been shown in [10], that these simple optimizations can improve over the original *Apriori* algorithm by orders of magnitude.

5.2.2 DepthProject Algorithm

The *DepthProject* algorithm is based on the notion of the lexicographic tree, defined in [5]. Unlike *TreeProjection*, the approach aggressively explores the candidates in a depth-first strategy both to ensure better pruning and faster counting. As in *TreeProjection*, the database is recursively projected down the lexicographic tree to ensure more efficient counting. This kind of projection ensures that the counting information for k-candidates is reused for $(k + 1)$-candidates, as in the case of *FP-growth*.

For the case of the *DepthProject* method [4], the lexicographic tree is explored in depth-first order to maximize the advantage of lookaheads in which entire subtrees can be pruned because it is known that all patterns in them are frequent. The overall pseudocode for the depth-first strategy is illustrated in Fig. 2.14. The pseudocodes for candidate generation and counting are not provided because they are similar to the previously discussed algorithms. However, one important distinction in counting is that projected databases are used for counting. This is similar to the *FP-growth* class of algorithms. Note that the recursive transaction projection is particularly effective with a depth-first strategy because a smaller number of projected databases need to be stored along a path in the tree, as compared to the breadth of the tree.

To reduce the overhead of counting long patterns, the notion of lookaheads are used. At any node P of the tree, let $F(P)$ be its possible (candidate) item extensions. Then, it is checked whether $P \cup F(P)$ is frequent in two ways:

1. Before counting the support of the individual extensions of P (i.e., $\{P \cup \{i\} : \forall i \in F(P)\}$), it is checked whether $P \cup F(P)$ occurs as subset of a frequent itemset that has already been discovered earlier during depth-first exploration. If such is the case, then the entire subtree rooted at P is pruned because it is known

to be frequent and it is not a maximal pattern. This type of pruning is particularly effective with a depth-first strategy.

2. During support counting of the item extensions, the support of $P \cup F(P)$ is also determined. If after support counting, $P \cup F(P)$ turns out to be frequent, then the entire subtree rooted at node P can be pruned. Note that the projected database at node P (as in *TreeProjection*) is used.

Although lookaheads are also used in the *MaxMiner* algorithm, it should be pointed out that the effectiveness of lookaheads is maximized with a depth-first strategy. This is true of the first of the two aforementioned strategies, in which it is checked whether $P \cup F(P)$ is a subset of an already existing frequent pattern. This is a because a depth-first strategy tends to explore the itemsets in dictionary order. In dictionary order, maximal itemsets are usually explored much earlier than *most* of their subsets. For example, for a 10-itemset $abcdefghij$, only 9 of the 1024 subsets of the itemsets will be explored before exploring the itemset $abscdefghij$. These 9 itemsets are the immediate prefixes of the itemset. When, the longer itemsets are explored early they become available to prune shorter itemsets.

The following information is stored at each node during the process of construction of the lexicographic tree:

1. The itemset P at that node.
2. The set of lexicographic tree extensions at that node which are $E(P)$.
3. A pointer to the projected transaction set $\mathcal{T}(Q)$, where Q is some ancestor of P (including itself). The root of the tree points to the entire transaction database.
4. A bitvector containing the information about which transactions contain the itemset for node P as a subset. The length of this bitvector is equal to the total number of transactions in $\mathcal{T}(Q)$. The value of a bit for a transaction is equal to one, if the itemset P is a subset of the transaction. Otherwise it is equal to zero. Thus, the number of 1 bits is equal to the number of transactions in $\mathcal{T}(Q)$ which project to P. The bitvectors are used to make the process of support counting more efficient.

After all the projected transactions at a given node have been identified, then finding the subtree rooted at that node is a completely independent itemset generation problem with a *substantially reduced* transaction set. The number of transactions at a node is proportional to the support at that node.

The description in Fig. 2.14 shows how the depth first creation of the lexicographic tree is performed. The algorithm is described recursively, so that the call from each node is a completely independent itemset generation problem that finds all frequent itemsets that are descendants of a node. There are three parameters to the algorithm, a pointer to the database $\mathcal{T}$, the itemset node N, and the bitvector B. The bitvector B contains one bit for each transaction in $T \in \mathcal{T}$, and indicates whether or not the transaction T should be used in finding the frequent extensions of N. A bit for a transaction T is one, if the itemset at that node is a subset of the corresponding transaction. The first call to the algorithm is from the *null* node, the parameter $\mathcal{T}$ is the entire transaction database. Because each transaction in the database is relevant to perform the counting, the bitvector B consists of all "one " values. One property

Fig. 2.14 The depth first strategy

```
Algorithm DepthFirst(Itemset Node: $N$,
    PointerToDatabase: $\mathcal{T}$, Bitvector : $B$)
  begin
  $C = GenerateCandidates(N)$;
  $E = Count(N, \mathcal{T}, B, C)$;
  { Let $E = \{i_1, \ldots, i_{|E|}\}$, in lexicographic order }
  Store frequent itemsets $N \cup \{i_r\}$ for $r \in \{1, \ldots, |E|\}$;
  $B' = CreateBitvector(N, B, \mathcal{T})$;
  if $(ProjectionCondition)$ then
    begin
    $\mathcal{T}' = Project(\mathcal{T}, E, N, B')$;
    Modify $B'$ to be a set of $|\mathcal{T}'|$ ones;
    end;
        else $\mathcal{T}' = \mathcal{T}$;
  for $r := 1$ to $|E|$ do $DepthFirst(N \cup \{i_r\}, \mathcal{T}', B')$;
  end

Subroutine Project(Database: $\mathcal{T}$,
      FrequentExtensions : $E$, Bitvector: $B$)
  begin
  $\mathcal{T}'$ = Empty set of transactions;
  for each transaction $T \in \mathcal{T}$ do
    begin
    if corresponding bit in $B$ is 1 then add $T \cap E$ to $\mathcal{T}'$;
    end
  return($\mathcal{T}'$);
  end

Subroutine CreateBitvector($N, B, \mathcal{T}$)
  begin
  Initialize $B' = B$;
  Let $n$ be the lexicographically largest item in $N$;
  for each transaction $T \in \mathcal{T}$ do
     if $n \notin T$ then set the corresponding bit in $B'$ to 0;
  return($B'$);
end
```

of the *DepthProject* algorithm is that the projection is performed only when the transaction database reduces by a certain size. This is the *ProjectionCondition* in Fig. 2.14.

Most of the nodes in the lexicographic tree correspond to the lower levels. Thus, the counting times at these levels account for most of the CPU times of the algorithm. For these levels, a strategy called bucketing can substantially improve the counting times. The idea is to change the counting technique at a node in the lexicographic tree, if $|E(P)|$ is less than a certain value. In this case, an upper bound on the number of distinct *projected* transactions is $2^{|E(P)|}$. Thus, for example, when $|E(P)|$ is nine, then there are only 512 distinct projected transactions at the node P. Clearly, this is because the projected database contains several repetitions of the same (projected)

Fig. 2.15 Aggregating bucket counts

```
Algorithm AggregateCounts($Counts{:}bucket[\ldots]$)
begin
  { We assume that there are $2^{|E(P)|}$ buckets, one
      corresponding to each bitstring of length $|E(P)|$ }
  $k = |E(P)|$;
  for $i := 1$ to $k$ do
    begin
    for $j := 1$ to $2^k$ do
    if the $i$th bit of bitstring representation
      of $j$ is 0 then
      begin
      $bucket[j] = bucket[j] + bucket[j + 2^{i-1}]$;
      end
    end
end
```

transaction. The fact that the number of *distinct* transactions in the projected database is small can be exploited to yield substantially more efficient counting algorithms. The aim is to count the support for the entire subtree rooted at P with a quick pass through the data, and an additional postprocessing phase which is independent of database size. The process of performing bucket counting consists of two phases:

1. In the first phase, the counts of each distinct transaction present in the projected database are determined. This can be accomplished easily by maintaining $2^{|E(P)|}$ buckets or counters, scanning the transactions one by one, and adding counts to the buckets. The time for performing this set of operations is linear in the number of (projected) database transactions.
2. In the second phase, the counts of the $2^{|E(P)|}$ transaction are used to determine the aggregate support counts for each itemset. In general, the support count of an itemset may be obtained by adding the counts of all the supersets of that itemset to it. A skillful algorithm (from the efficiency perspective) for performing these operations is illustrated in Fig. 2.15.

Consider a string composed of 0, 1, and $*$ that refers to an itemset in which the positions with 0 and 1 are fixed to those values (corresponding to presence or absence of items), while a position with a $*$ is a "don't care". Thus, all itemsets can be expressed in terms of 1 and $*$ because itemsets are traditionally defined with respect to presence of items. Consider for example, the case when $|E(P)| = 4$, and there are four items, numbered $\{1, 2, 3, 4\}$. An itemset containing items 2 and 4 is denoted by $*1*1$. We start off with the information on $2^4 = 16$ bitstrings which are composed of 0 and 1. These represent all possible distinct transactions. The algorithm aggregates the counts in $|E(P)|$ iterations. The count for a string with a "*" in a particular position may be obtained by adding the counts for the strings with a 0 and 1 in those positions. For example, the count for the string *1*1 may be expressed as the sum of the counts of the strings 01*1 and 11*1.

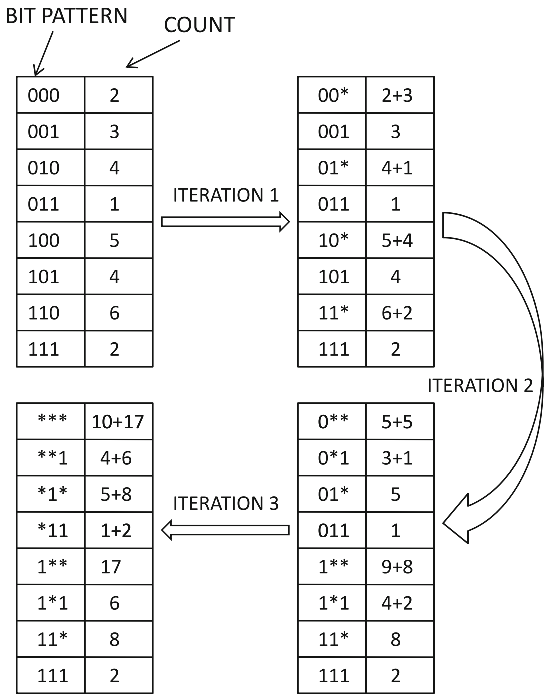

Fig. 2.16 Performing the second phase of bucketing

The procedure in Fig. 2.15 works by starting with the counts of the 0–1 strings, and then converts them to strings with 1 and *. The algorithm requires $|E(P)|$ iterations. In the ith iteration, it increases the counts of all those buckets with a 0 in the ith bit, so that the count now corresponds to a case when that bucket contains a $*$ in that position. This can be achieved by adding the counts of the buckets with a 0 in the ith position to that of the bucket with a 1 in that position, with all other bits having the same value. For example, the count of the string 0*1* is obtained by adding the counts of the buckets 001* and 011*. In Fig. 2.15, the process of adding the count of the bucket j to that of the bucket $j + 2^{i-1}$ achieves this.

The second phase of the bucketing operation requires $|E(P)|$ iterations, and each iteration requires $2^{|E(P)|}$ operations. Therefore, the total time required by the method is proportional to $2^{|E(P)|} \cdot |E(P)|$. When $|E(P)|$ is sufficiently small, the time required by the second phase of postprocessing is small compared to the first phase, whereas the first phase is essentially proportional to reading the database for the current projection.

We have illustrated the second phase of bucketing by an example in which $|E(P)| = 3$. The process illustrated in Fig. 2.16 illustrates how the second phase of bucketing is efficiently performed. The exact strings and the corresponding counts in each of the $|E(P)| = 3$ iterations are illustrated. In the first iteration, all those bits with 0 in the lowest order position have their counts added with the count of the bitstring with a 1 in that position. Thus, $2^{|E(P)|-1}$ pairwise addition operations take place during this step. The same process is repeated two more times with the second and third order bits. At the end of three passes, each bucket contains the support count for the appropriate itemset, where the '0' for the itemset is replaced by a "don't care" which is represented by a '*'. Note that the number of transactions in this example is 27. This is represented by the entry for the bucket ***. Only two transactions contain all three items that is represented by the bucket 111.

The projection-based methods were shown to have an order of magnitude improvement over the *MaxMiner* algorithm. The depth-first approach has subsequently been used in the context of many tree-based algorithms. Other examples of such algorithms include those in [17, 18, 14]. Among these, the MAFIA algorithm [14] is discussed in some detail in the next subsection. An approach which varies on the projection methodology, and uses *opportunistic projection* is discussed in [38]. This algorithm opportunistically chooses between array-based and tree-based representations to represent projected transaction subsets. Such an approach has been shown to be more efficient than many state of the art methods such as the FP-Growth method. Other variations of tree-based algorithms have also been proposed [70] that use different strategies in tree exploration.

5.2.3 MAFIA Algorithm

The MAFIA algorithm proposed in [14] shares a number of similarities to the *Depth-Project* approach, though it uses a bitmap based approach for counting, rather than the use of a projected transaction database. In the bitmap-based approach, a sequence of bits is maintained for each itemset that corresponds to whether or not that transaction contains that particular item. Sparse representations (such as a list of transaction identifiers) may also be used, when the fraction of transactions containing the itemset is small. Note that such an approach may be considered a special case of database projection [5], in which vertical projection is used but horizontal projection is not. This has the advantage of requiring less memory, but it reuses a smaller fraction of the counting information from higher level nodes. A number of other pruning optimizations have also been proposed in this work that further improve the effectiveness of the algorithm. In particular, it has been pointed out that when the support of the extension of a node is the same as that of its parent, then that subtree can be pruned away, because of the counts of all the itemsets in the subtree can be derived from those of other itemsets in the data. This is the same as the support lower bounding trick discussed in Sect. 2.4, and also used in *MaxMiner* for pruning. Thus, the approach in [14] uses many of the same strategies used in *MaxMiner* and *TreeProjection*, but with in a different combination, and with some variations on specific implementation details.

5.2.4 GenMax

Like MAFIA, *GenMax* is a uses the vertical representation to speed up counting. Specifically the *tidlists* are used by *GenMax* to speed up the counting approach. In particular the more recent notion of *diffsets* [72] was used, and a depth-first exploration strategy was used. An approach known as successive focussing was used to further improve the efficiency of the algorithm. The details of the *GenMax* approach may be found in [28].

5.3 Frequent Closed Itemset Mining Algorithms

The are several frequent closed itemset mining algorithms [41, 42, 51–53, 64, 66–69, 73] exist to date. Most of the maximal and closed pattern mining algorithms are based on different variations of the non-maximal pattern mining algorithms. Typically pruning strategies are incorporated within the non-maximal pattern mining algorithms to yield more efficient algorithms.

5.3.1 Close

In this algorithm [52] authors apply *Apriori* based patten generation over the closed itemset search space. The usages of closed itemset lattice (search space) significantly reduces the overall search space of the algorithm. *Close* operates in iterative manner. Each iteration consists of three phases, . First, the closure function is applied for obtaining the candidate closed itemsets and their support. Next, the obtained set of candidate closed itemsets are tested against the minimum support constraint. If succeed, the candidates are marked as frequent closed itemset. Finally the same procedure is initiated to generate the next level of candidate closed itemsets. This process continues until all frequent closed itemsets have been generated.

5.3.2 CHARM

CHARM [73] is a frequent closed itemset mining algorithm, that takes advantage of the vertical representation of database as in the case of *Eclat* [71] for efficient closure checking operation. For punning the search space *CHARM* uses the following three properties. Suppose for itemset P and Q, if tidset(P) = tidset(Q), then it replaces every occurrence of P by $P \cup Q$ and prune the whole branch under Q. On the other hand if tidset(P) $\subset$ tidset(Q), it replaces every occurrence of P by $P \cup Q$, but does not prune the branch under Q. Finally if, tidset(P)<>tidset(Q), none of the aforementioned prunings can be applied. The initial call of *CHARM* accepts a set(I) of single length frequent item and minimum support as input. As a first step, it sorts I by the increasing the order of support of the items. For each item P, *CHARM* tries to extend it by another item Q from the same set and applies three conditions for pruning. If the newly create itemset by extension is frequent, *CHARM* performs closure-checking to identify whether the itemset is closed. *CHARM* also updates the set I accordingly. In other words, it replaces P with $P \cup Q$, if the corresponding pruning condition is met. If the set I is the not empty, then *CHARM* is called recursively.

5.3.3 CLOSET and CLOSET+

CLOSET [53] and *CLOSET+* [69] frequent closed itemset mining algorithms are inspired by the *FP-growth* method. The *CLOSET* algorithm makes use of the principles of the FP-Tree data structure to avoid the candidate generation step during the process of mining frequent closed itemsets. This work introduces a technique, referred to as single prefix path compression, that quickly assists the mining process. *CLOSET* also applies partition-based projection mechanisms for better scalability. The mining procedure of *CLOSET* follows the *FP-growth* algorithm. However, the algorithm is able to extract only the closed patterns by careful book-keeping. *CLOSET* treats items appearing in every transaction of the conditional database specially. For example, if Q is the set of items that appear in every transaction of the P conditional database then $P \cup Q$ creates a frequent closed itemset if it is not a proper subset of any frequent closed itemset with the equal support. *CLOSET* also prunes the search space. For example, if P and Q are frequent itemset with the equal support where Q is also a closed itemset and $P \subset Q$, then it does not mine the conditional database of P because the latter will not produce any frequent closed itemsets.

CLOSET+ is a follow-up work after *CLOSET* by the same group of authors. *CLOSET+* attempts to design the most optimized frequent closed itemset mining algorithm by finding the best trade-off between depth-first search versus breadth-first search, vertical formats versus horizontal formats, tree structure versus other data structures, top–down versus bottom–up traversal, and pseudo projection versus physical projection of the conditional database. *CLOSET+* keeps track of the unpromising prefix itemsets for generating potential closed frequent itemsets and prunes the search space by deleting them. *CLOSET+* also applies "item merging," and "sub-itemset" based pruning. To save the memory of the closure checking operation, *CLOSET+* uses the combination of the 2-level hash-indexed tree based method and the pseudo-projection based upward checking method. Interested readers are encouraged to refer to [69] for more details.

5.3.4 DCI_CLOSED

DCI_CLOSED [41, 42] uses a bitwise vertical representation of the input database. *DCI_CLOSED* can be executed independently on each partition of the database in any order and, thus, also in parallel. *DCI_CLOSED* is designed to improve memory-efficiency by avoiding the storage of duplicate closed itemsets. *DCI_CLOSED* designs a novel strategy for searching the lattice that can detect and discard duplicate closed patterns on the fly. Using the concept of order-preserving generators of frequent closed itemsets, a new visitation scheme of the search space is introduced. Such a visitation scheme results a disjoint sub division of the search space. This also facilitates parallelism.*DCI_CLOSED* applies several optimization tricks to improve execution time, such as the bitwise intersection of tidsets to compute support and closure. Where possible, it reuses previously computed intersections to avoid redundant computations.

6 Other Optimizations and Variations

In this section, a number of other optimizations and variations of frequent pattern mining algorithms will be discussed. Many of these methods are discussed in detail in other chapters of this book, and therefore they will be discussed only briefly here.

6.1 Row Enumeration Methods

Not all frequent pattern mining algorithms follow the fundamental steps of baseline algorithm, there exists a number of special cases, for which specialized frequent pattern mining algorithms have been designed. An interesting case is that of micro-array data sets, in which the columns are very long but the number of rows are not very large. In such cases, a method called *row-enumeration* is used [22, 23, 40, 48, 49] instead of the usual column enumeration, in which combinations of rows are examined during the search process. There are two categories of row enumeration algorithm. One category algorithm perform bottom-up [22, 23, 48] search over the row enumeration tree whereas other category algorithms perform top-down[40] search strategy.

Row enumeration algorithms perform mining over the transpose of the transaction database. In transpose database, each transaction id become item and each item corresponds a transaction. Mining over the transposed database is basically the bottom up search for frequent patterns by enumeration of row sets. However, the bottom-up search strategy cannot take advantage of user-specified minimum support threshold to effectively prune the search space, and therefore leads to longer running time and large memory overhead. As a solution [40] introduce a top-down approach of mining using a novel row enumeration tree. Their approach can take full advantage of user-defined minimum support value and prune the search space efficiently hence lower down the execution time.

Note that, both of the search strategies are applied over the transposed transaction database. Most of developed algorithm using row enumeration technique concentrate on mining frequent closed itemset (explained in Sect. 5). The reason behind this motivation is that due to the nature of micro-array data there exists a large number of redundancy among the frequent patterns for a minimum support threshold and closed patterns are capable of summarizing the whole database. These strategies will be discussed in detail in Chap. 4, and therefore only a brief discussion is provided here.

6.2 Other Exploration Strategies

The advantage of tree-enumeration strategies is that they facilitate the exploration of candidates in the tree in an arbitrary order. A method known as *Pincer-Search* is proposed in [37] that combines top-down and bottom-up exploration in "pincer" fashion to avail of the advantages of both subset and superset pruning. Two primary observations are used in pincer search:

1. Any subset of a frequent itemset is frequent.
2. Any superset of an infrequent itemset is infrequent.

In pincer-search, top–down and bottom–up exploration are combined and irrelevant itemsets are pruned using both observations. More details of this approach are discussed in [37]. Note that, for sparse transaction data, superset pruning is likely to be inefficient. Other recent methods have been proposed for long pattern mining with methods such as "leap search." These methods are discussed in the chapter on long pattern mining in this book.

7 Reducing the Number of Passes

A major challenge in frequent pattern mining is when the data is disk resident. In such cases, it is desirable to use level-wise methods to ensure that random accesses to disk are minimized. This is the reason that most of the available algorithms use level-wise methods, which ensure that the number of passes over the database are bounded by the size of the longest pattern. Even so, this can be significant, when many long patterns are present in the database. Therefore, a number of methods have been proposed in the literature to reduce the number of passes over the data. These methods could be used in the context of join-based algorithms, tree-based algorithms, or even other classes of frequent pattern mining methods. These correspond to combining the level-wise database passes, using sampling, and using a preprocess-once-query-many paradigm.

7.1 Combining Passes

The earliest work on combining passes was proposed in the original *Apriori* algorithm [1]. The key idea in combing passes is that it is possible to use joins to create candidates of higher order than $(k+1)$ in a single pass. For example, $(k+2)$-candidates can be created from $(k+1)$-candidates before actual validation of the $(k+1)$-candidates over the data. Then, the candidates of size $(k+1)$ and $(k+2)$ can be validated together in a single pass over the data. Although such an approach reduces the number of passes over the data, it has the downside that the number of spurious $(k+2)$ candidates will be far larger because the $(k+1)$ candidates were not confirmed to be frequent before they were joined. Therefore, the saving of database passes comes at an increased computational cost. Therefore, it was proposed in [1] that the approach should be used for later passes, when the number of candidates has already reduced significantly. This reduces the likelihood that the number of candidates blows up too much with this approach.

7.2 Sampling Tricks

A number of sampling tricks can be used to greatly improve the efficiency of the frequent pattern mining process. Most sampling methods require two passes over the data, the first of which is used for sampling. An interesting approach that uses two passes with the use of sampling is discussed in [65]. This method generates the approximately frequent patterns over the data, using a sample. False negatives can be reduced by lowering the minimum support level appropriately, so that bounds can be defined on the likelihood of false negatives. False positives can be removed with the use of a second pass over the data. The major downside of the approach is that the reduction in the minimum support level to reduce the number of false negatives can be significant. This also reduces the computational efficiency of the approach. The method however requires only two passes over the data, where the first pass is used to create the sample, and the second pass is used to remove the false positives.

An interesting approach proposed in [57] divides the disk resident database into smaller memory-resident partitions. For each partition, more efficiency algorithms can be used, because of the memory-resident nature of the partition. It should be pointed out that each frequent pattern over the entire database will appear as a frequent pattern in at least one transaction. Therefore, the union of the itemsets over the different transactions provides a superset of the true frequent patterns. A post-processing phase is then used to filter out the spurious itemsets, by counting this candidate set against the transaction database. As long as the partitions are reasonably large, the superset found approximates the true frequent patterns very well, and therefore the additional time spent in counting irrelevant candidates is relatively small. The main advantage of this approach is it requires only two passes over the database. Therefore, such an approach is particularly effective when the data is resident on disk.

The *Dynamic Itemset Counting (DIC)* algorithm [15] divides the database into intervals, and generates longer candidates when it is known that the subsets of these candidates are already frequent. These are then validated over the database. Such an approach can reduce the number of passes over the data, because it implicitly combines the process of candidate generation and counting.

7.3 Online Association Rule Mining

In many applications, a user may wish to query the transaction data to find the association rules or the frequent patterns. In such cases, even at high support levels, it is often impossible to create the frequent patterns in online time because of the multiple passes required over a potentially large database. One of the earliest algorithms for online association rule mining was proposed in [6]. In this approach, an augmented lexicographic tree is stored either on disk or in main-memory. The lexicographic tree is augmented with all the edges represented the subset relationships between itemsets, and is also referred to as the itemset *lattice*. For any given query, the itemset lattice may be traversed to determine the association rules. It has been shown in [6], that such an approach can also be used to determine the non-redundant association rules in the underlying data. A second method [40] uses a condensed frequent pattern tree (instead of a lattice) to pre-process and store the itemsets. This structure can be queried to provide online responses.

A very different approach for online association rule mining has been proposed in [34], in which the transaction database is processed in real time. In this case, an incremental approach is used to mine the transaction database. This is a *Continuous Association Rule Mining Algorithm*, which is referred to as *CARMA*. In this case, transactions are processed as they arrive, and candidate itemsets are generated on the fly, by examining the subsets of that transaction. Clearly, the downside is that such an approach is that it will create a lot more candidates than any of the offline algorithms which use levelwise methods to generate the candidates. This general characteristic is of course true of any algorithm which tries to reduce the number of passes with approximate candidate generation. One interesting characteristic of the CARMA algorithm is that it allows the user to change the minimum support level during execution. In that case, the algorithm is guaranteed to have generated the supersets of the true itemsets in the data. If desired, a second pass over the data can be used to remove the spurious frequent itemsets.

Many streaming methods have also been proposed that use only one pass over the transaction data [19–21, 35, 43]. It should be pointed out that it is often difficult to find even 1-itemsets exactly over a data stream because of the one-pass constraint [21], when the number of distinct items is larger than the main memory availability. This is often true of k-itemsets as well, especially at low support levels. Furthermore, if the patterns in the stream change over time, then the frequent k-itemsets will change significantly as well. These methods therefore have the challenge of finding the frequent itemsets efficiently, maintaining them, and handling issues involving evolution of the data stream. Given the numerous challenges of pattern mining in this scenario, most of these methods find the frequent items approximately. These issues will be discussed in detail in Chap. 9 on streaming pattern mining algorithms.

8 Conclusions and Summary

This chapter provides a survey of different frequent pattern mining algorithms. most frequent pattern algorithms, implicitly or explicitly, explore the enumeration tree of itemsets. Algorithms such as *Apriori* explore the enumeration tree in breadth-first fashion with join-based candidate generation. Although the notion of an enumeration tree is not explicitly mentioned by the *Apriori* algorithm, the execution tree explores the candidates according to an enumeration tree constructed on the prefixes. Other algorithms such as *TreeProjection* and *FP-growth* use the hierarchical relationships between the projected databases for patterns of different lengths, and avoid re-doing the counting work done for the shorter patterns. Maximal and closed versions of frequent pattern mining algorithms are also able to achieve much better pruning performance. A number of efficiency-based optimizations of frequent pattern mining algorithms were also discussed in this chapter.

References

1. R. Agrawal, and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases, *VLDB Conference*, pp. 487–499, 1994.
2. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Conference*, 1993.
3. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules, *Advances in Knowledge Discovery and Data Mining*, pp. 307–328, 1996.
4. R. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth-first Generation of Long Patterns, *ACM KDD Conference*, 2000. Also available as IBM Research Report, RC21538, July 1999.
5. R. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets, *Journal of Parallel and Distributed Computing*, 61(3), pp. 350–371, 2001. Also available as IBM Research Report, RC21341, 1999.
6. C. C. Aggarwal, P. S. Yu. Online Generation of Association Rules, *ICDE Conference*, 1998.
7. C. C. Aggarwal, P. S. Yu. A New Framework for Itemset Generation, *ACM PODS Conference*, 1998.
8. E. Azkural and C. Aykanat. A Space Optimization for FP-Growth, *FIMI workshop*, 2004.
9. Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining Frequent Patterns with Counting Inference. *ACM SIGKDD Explorations Newsletter*, 2(2), pp. 66–75, 2000.
10. R. J. Bayardo Jr. Efficiently mining long patterns from databases, *ACM SIGMOD Conference*, 1998.
11. J. Blanchard, F. Guillet, R. Gras, and H. Briand. Using Information-theoretic Measures to Assess Association Rule Interestingness. *ICDM Conference*, 2005.
12. C. Borgelt, R. Kruse. Induction of Association Rules: Apriori Implementation, *Conference on Computational Statistics*, 2002. http://fuzzy.cs.uni-magdeburg.de/ borgelt/software. html.
13. J.-F. Boulicaut, A. Bykowski, and C. Rigotti. Free-sets: A Condensed Representation of Boolean data for the Approximation of Frequency Queries. *Data Mining and Knowledge Discovery*, 7(1), pp. 5–22, 2003.
14. D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases, *ICDE Conference*, 2000. Implementation URL: http://himalaya-tools.sourceforge.net/Mafia/.
15. S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *ACM SIGMOD Conference*, 1997.

16. S. Brin, R. Motwani, and C. Silverstein. Beyond Market Baskets: Generalizing Association Rules to Correlations. *ACM SIGMOD Conference*, 1997.
17. T. Calders, and B. Goethals. Mining all non-derivable frequent itemsets *Principles of Data Mining and Knowledge Discovery*, pp. 1–42, 2002.
18. T. Calders, and B. Goethals. Depth-first Non-derivable Itemset Mining, *SDM Conference*, 2005.
19. T. Calders, N. Dexters, J. Gillis, and B. Goethals. Mining Frequent Itemsets in a Stream, *Informations Systems*, to appear, 2013.
20. J. H. Chang, and W. S. Lee. Finding Recent Frequent Itemsets Adaptively over Online Data Streams, *ACM KDD Conference*, 2003.
21. M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Automata, Languages and Programming*, pp. 693–703, 2002.
22. G. Cong, A. K. H. Tung, X. Xu, F. Pan, and J. Yang. FARMER: Finding interesting rule groups in microarray datasets. *ACM SIGMOD Conference*, 2004.
23. G. Cong, K.-L. Tan, A. K. H. Tung, X. Xu. Mining Top-*k* covering Rule Groups for Gene Expression Data. *ACM SIGMOD Conference*, 2005.
24. M. El-Hajj and O. Zaiane. COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation. *FIMI Workshop*, 2003.
25. F. Geerts, B. Goethals, J. Bussche. A Tight Upper Bound on the Number of Candidate Patterns, *ICDM Conference*, 2001.
26. B. Goethals. Survey on frequent pattern mining, *Technical report, University of Helsinki*, 2003.
27. R. P. Gopalan and Y. G. Sucahyo. High Performance Frequent Pattern Extraction using Compressed FP-Trees, *Proceedings of SIAM International Workshop on High Performance and Distributed Mining*, 2004.
28. K. Gouda, and M. Zaki. Genmax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3), pp. 223–242, 2005.
29. G. Grahne, and J. Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets, *IEEE ICDM Workshop on Frequent Itemset Mining*, 2004.
30. G. Grahne, and J. Zhu. Fast Algorithms for Frequent Itemset Mining Using FP-Trees. *IEEE Transactions on Knowledge and Data Engineering*. 17(10), pp. 1347–1362, 2005, vol. 17, no. 10, pp. 1347–1362, October, 2005.
31. V. Guralnik, and G. Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Computing*, 30(4): pp. 443–472, April 2004.
32. J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation, *ACM SIGMOD Conference*, 2000.
33. J. Han, H. Cheng, D. Xin, and X. Yan. Frequent Pattern Mining: Current Status and Future Directions, *Data Mining and Knowledge Discovery*, 15(1), pp. 55–86, 2007.
34. C. Hidber. Online Association Rule Mining, *ACM SIGMOD Conference*, 1999.
35. R. Jin, and G. Agrawal. An Algorithm for in-core Frequent Itemset Mining on Streaming Data, *ICDM Conference*, 2005.
36. Q. Lan, D. Zhang, and B. Wu. A New Algorithm For Frequent Itemsets Mining Based On Apriori And FP-Tree, *IEEE International Conference on Global Congress on Intelligent Systems*, pp. 360–364, 2009.
37. D.-I. Lin, and Z. Kedem. Pincer-search: A New Algorithm for Discovering the Maximum Frequent Set, *EDBT Conference*, 1998.
38. J. Liu, Y. Pan, K. Wang. Mining Frequent Item Sets by Opportunistic Projection, *ACM KDD Conference*, 2002.
39. G. Liu, H. Lu and J. X. Yu. AFOPT:An Efficient Implementation of Pattern Growth Approach, *FIMI Workshop*, 2003.
40. H. Liu, J. Han, D. Xin, and Z. Shao. Mining frequent patterns on very high dimensional data: a top- down row enumeration approach. *SDM Conference*, 2006.
41. C. Lucchesse, S. Orlando, and R. Perego. DCI-Closed: A fast and memory efficient algorithm to mine frequent closed itemsets. *FIMI Workshop*, 2004.

42. C. Lucchese, S. Orlando, and R. Perego. Fast and memory efficient mining of frequent closed itemsets. *IEEE TKDE Journal*, 18(1), pp. 21–36, January 2006.
43. G. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. *VLDB Conference*, 2002.
44. H. Mannila, H. Toivonen, and A.I. Verkamo. Efficient algorithms for discovering association rules. *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pp. 181–192, 1994.
45. B. Negrevergne, T. Guns, A. Dries, and S. Nijssen. Dominance Programming for Itemset Mining. *IEEE ICDM Conference*, 2013.
46. S. Orlando, P. Palmerini, R. Perego. Enhancing the a-priori algorithm for frequent set counting, *Third International Conference on Data Warehousing and Knowledge Discovery*, 2001.
47. S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. *ICDM Conference*, 2002.
48. F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki. Finding closed patterns in long biological datasets. *ACM KDD Conference*, 2003.
49. F Pan, A. K. H. Tung, G. Cong, X. Xu. COBBLER: Combining column and Row Enumeration for Closed Pattern Discovery. *SSDBM*, 2004.
50. J.-S. Park, M. S. Chen, and P. S. Yu. An Effective Hash-based Algorithm for Mining Association Rules, *ACM SIGMOD Conference*, 1995.
51. N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *ICDT Conference*, 1999.
52. N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Journal of Information Systems*, 24(1), pp. 25–46, 1999.
53. J. Pei, J. Han, and R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets, *DMKD Workshop*, 2000.
54. J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases, *ICDM Conference*, 2001.
55. B. Racz. nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree, *FIMI Workshop*, 2004.
56. M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A Perspective on Databases and Data Mining, *ACM KDD Conference*, 1995.
57. A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. *VLDB Conference*, 1995.
58. P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, D. Shah. Turbo-charging Vertical Mining of Large Databases. *ACM SIGMOD Conference*, pp. 22–33, 2000.
59. Z. Shi, and Q. He. Efficiently Mining Frequent Itemsets with Compact FP-Tree, IFIP International Federation for Information Processing, V-163, pp. 397–406, 2005.
60. R. Srikant. Fast algorithms for mining association rules and sequential patterns. *PhD thesis, University of Wisconsin, Madison*, 1996.
61. Y. G. Sucahyo and R. P. Gopalan. CT-ITL: Efficient Frequent Item Set Mining Using a Compressed Prefix Tree with PatternGrowth, *Proceedings of the 14th Australasian Database Conference*, 2003.
62. Y. G. Sucahyo and R. P. Gopalan. CT-PRO: A Bottom Up Non Recursive Frequent Itemset Mining Algorithm Using Compressed FP-Tree Data Structures. *FIMI Workshop*, 2004.
63. P.-N. Tan, V. Kumar, amd J. Srivastava. Selecting the Right Interestingness Measure for Association Patterns. *ACM KDD Conference*, 2002.
64. I. Taouil, N. Pasquier, Y. Bastide, and L. Lakhal. Mining Basis for Association Rules using Closed Sets, *ICDE Conference*, 2000.
65. H. Toivonen. Sampling large databases for association rules. *VLDB Conference*, 1996.
66. T. Uno, M. Kiyomi and H. Arimura. Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets, *FIMI Workshop*, 2004.
67. J. Wang, J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. *ICDE Conference*, 2004.

68. J. Wang, J. Han, Y. Lu, and P. Tzvetkov. TFP: An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Transactions on Knowledge and Data Engineering*, 17, pp. 652–664, 2002.
69. J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the Best strategies for mining frequent closed itemsets. *ACM KDD Conference*, 2003.
70. G. I. Webb. Efficient Search for Association Rules, *ACM KDD Conference*, 2000.
71. M. J. Zaki. Scalable algorithms for association mining, *IEEE Transactions on Knowledge and Data Engineering*, 12(3), pp. 372–390, 2000.
72. M. Zaki, and K. Gouda. Fast vertical mining using diffsets. *ACM KDD Conference*, 2003.
73. M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. *SDM Conference*, 2002.
74. M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *KDD Conference*, pp. 283–286, 1997.
75. C. Zeng, J. F. Naughton, and JY Cai. On Differentially Private Frequent Itemset Mining. In Proceedings of 39th International Conference on Very Large data Bases, 2012.