18
Association Rule Mining III: Frequent Pattern Trees

18.1 Introduction: FP-Growth

The *Apriori* algorithm described in Chapter 17 is a successful method of deriving association rules from a transaction database. However it has important shortcomings. In this chapter an alternative method, known as the *FP-growth algorithm* is presented, which aims to overcome these. Before expanding on this, we will start by recapping on some of the basic points from Chapter 17.

It is assumed that we have a database of *transactions*, each comprising a number of *items*, such as

milk, fish, cheese
eggs, milk, pork, butter
cheese, cream, bread, milk, fish

Each record corresponds to a transaction such as one person's purchases in a supermarket. A collection of items, such as $\{fish, pork, cream\}$ is known as an *itemset*.

The *support count* (or just *count*) of an itemset is the number of times that the items occur together in a transaction, possibly with other items. Thus for the above database of three transactions $count(\{milk\}) = 3$, $count(\{pork\}) = 1$, $count(\{cheese, milk\}) = 2$, $count(\{fish, milk\}) = 2$ etc.

The *support* of an itemset is defined as the value of the support count divided by the number of transactions in the database.

M. Bramer, *Principles of Data Mining*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-7307-6_18

The aim is to find *association rules* linking the items in purchases together, e.g.

eggs, **milk** → **bread**, **cheese**, **pork**

meaning that transactions that contain eggs and milk generally also include bread, cheese and pork.

We do this in two stages:

1. Find *itemsets* such as $\{eggs, milk, bread\}$ with a sufficiently high value of *support* (defined by the user).
2. For each such itemset, extract one or more association rules, with all the items in the itemset appearing on either the left- or the right-hand side.

This chapter is only concerned with step (1) of this process, i.e. finding the itemsets. A method for extracting the association rules from the itemsets is described in Section 17.8 of Chapter 17.

The term used in Chapter 17 for itemsets with a sufficiently high value of *support* was *supported itemsets*. In view of the title of this chapter we will switch here to using the equivalent term *frequent itemsets*, which is more commonly used in the technical literature, although perhaps less meaningful. (We will use the term frequent itemsets rather than frequent patterns.)

There is another detailed change from Chapter 17. In that chapter the definition of a frequent (or supported) itemset was that the value of the support count divided by the number of transactions in the database, i.e. the *support*, was greater than or equal to a threshold value defined by the user, such as 0.01, called *minsup*. This is equivalent to saying that the support count must be greater than or equal to the number of transactions multiplied by the value of *minsup*. For a database with a million transactions the value of *minsup* multiplied by the number of transactions would be a large number such as 10,000.

> In this chapter we will define a *frequent itemset* as one for which the support count is greater than or equal to a user-defined integer which we will call *minsupportcount*.

These two definitions are clearly equivalent. The value of *minsupportcount* will typically be a large integer, but for the example used in the remainder of this chapter we will set it to the highly unrealistic value of three.

An important result which was established in Chapter 17 is the *downward closure* property of itemsets: if an itemset is frequent, any (non-empty) subset of it is also frequent. This is generally used in a different form: if an itemset is infrequent then any superset of it must also be infrequent. For example if $\{a, b, c, d\}$ is infrequent then $\{a, b, c, d, e, f\}$ must also be infrequent. If the

latter were frequent, then $\{a, b, c, d\}$ as a subset of it must also be frequent, but we know that it is not. The practical significance of this result is that the only itemsets with, say, 6 elements that are worth considering are those that are created from a frequent itemset with 5 elements by adding an additional item.

We now return to the *Apriori* algorithm. Although very effective, it suffers from two disadvantages.

- The number of candidate itemsets to be considered can be very large, especially those with two elements. If there are n single-item itemsets, e.g. $\{fish\}$ that are frequent, the number of two-item itemsets generated for examination will be approximately $n^2/2$. As n might easily be tens of thousands this is a lot of itemsets to process, the large majority of which are likely to prove infrequent.
- Even though *Apriori* reduces the number of database scans considerably compared with more primitive methods, the number of scans can still be substantial and this can place a large processing overhead on the system, especially for large transaction databases.

One of the most popular alternative approaches to generating association rules is the *FP-Growth* (standing for *Frequent Pattern Growth*) algorithm, which was introduced by Han et al. [1]. The aim is to find all the frequent itemsets that can be extracted from the transaction database as efficiently as possible. One way of improving on the efficiency of the *Apriori* algorithm is to reduce the number of database scans. Another is to examine as few of the infrequent itemsets as possible. The number of possible (non-empty) itemsets for a database with n different items is $2^n - 1$, of which only a relatively small number are likely to be frequent, so reducing the number of infrequent ones examined is very important. Even for the very small transaction database with just three items shown above there are 8 different items, giving $2^8 - 1 = 255$ possible itemsets. For even quite a small supermarket the number of items could easily be several thousand.

The *FP-growth* algorithm has two stages.

- First the transaction database is processed to produce a data structure called a *FP-tree* (Frequent Pattern Tree) which captures the essence of the database as far as extracting frequent itemsets is concerned.
- Next the FP-tree is processed recursively, by constructing a sequence of reduced trees known as *conditional FP-trees*.

The transaction database is only processed at the first of these stages and is only scanned twice. As for virtually any conceivable alternative method the

database would have to be scanned at least once, reducing the number of scans to just two is a very valuable feature of this algorithm.

In [1] it is claimed that *FP-growth* is an order of magnitude faster than *Apriori*. Naturally this depends on a number of factors, for example whether the FP-tree can be represented in a way that is compact enough to fit into main memory. Like virtually all the algorithms in this book, there are a number of variants of both *Apriori* and *FP-growth* that aim to make them less memory or computationally expensive and there will no doubt be more in the future.

In the following sections the *FP-growth* algorithm is described and illustrated by a series of figures showing the FP-tree corresponding to an example transaction database, followed by a sequence of conditional FP-trees from which it is straightforward to extract the frequent itemsets.

18.2 Constructing the FP-tree

18.2.1 Pre-processing the Transaction Database

To illustrate the process we will use the transaction data from [1]. There are just five transactions held in a transaction database, with each item represented by a single letter:

f, a, c, d, g, i, m, p
a, b, c, f, l, m, o
b, f, h, j, o
b, c, k, s, p
a, f, c, e, l, p, m, n

The first step is to make a scan through the transaction database to count the number of occurrences of each item, which is the same as the support count of the corresponding single-item itemset. The result is as follows.

f, c: 4
a, m, p, b: 3
l, o: 2
d, g, i, h, j, k, s, e and n: 1

The user now needs to decide on a value for *minsupportcount*. As the amount of data is so small, in this example we will use the highly unrealistic value: ***minsupportcount*** $= \mathbf{3}$.

There are only six items for which the corresponding single-item itemset has a support count of *minsupportcount* or more. In descending order of sup-

port count they are: *f*, *c*, *a*, *b*, *m* and *p*. We store them in an array named *orderedItems* (Figure 18.1).

index	*orderedItems*
0	f
1	c
2	a
3	b
4	m
5	p

Figure 18.1 *orderedItems* array

As far as extracting frequent itemsets is concerned the items that are not in the *orderedItems* array may as well not exist, as they cannot occur in any frequent itemset. For example, if item *g* were a member of a frequent itemset then by the downward closure property of itemsets any non-empty subset of that itemset would also be frequent, so $\{g\}$ would have to be frequent, but we know by counting that it is not.

It is conventional and very important from a computational point of view that the items in an itemset are written in a fixed order. In the case of *FP-growth* they are written in descending order of their position in the *orderedItems* array, i.e. in descending order of the number of transactions in which each of them occurs. Thus $\{c, a, m\}$ is a valid itemset, which may be frequent or infrequent, but $\{m, c, a\}$ and $\{c, m, a\}$ are invalid. We are only interested in whether itemsets that are valid in this sense are frequent or infrequent.

We next make the second and final scan through the transaction database. As each transaction is read all items that are not in *orderedItems* are removed and the remaining items are sorted into descending order (i.e. the order of the items in *orderedItems*) before being passed to the FP-tree construction process.

This gives the same effect as if the transaction data were originally the five transactions

f, c, a, m, p
f, c, a, b, m
f, b
c, b, p
f, c, a, m, p

but the transaction database itself is left unchanged.

We now go on to describe the process of creating the FP-tree and extracting frequent itemsets from it. Although the transaction data is taken from [1] this description and especially the method of representing the evolving trees by arrays is the current author's own and the responsibility for any accidental errors or distortions is his alone.

18.2.2 Initialisation

Diagrammatically we can represent the initial state of the FP-tree by a single node, representing the root.

We will also represent the evolving tree by the contents of four arrays:

- Two two-dimensional arrays *nodes* and *child*, with a numerical index that will correspond to the numbering of the nodes in the tree (zero indicates the root node). The names given to the columns of these arrays are shown in Figure 18.2. Note that *child* can have an indefinite number of columns, but only the first two are needed for this example.
- Single-dimensional arrays *startlink* and *endlink* indexed by the names of the itemsets in the *orderedItems* array, i.e. *f*, *c*, *a*, *b*, *m* and *p*.

index	*item name*	*count*	*linkto*	*parent*
0	*root*			

nodes array

child1	*child2*

child array

index	*startlink*	*endlink*
f		
c		
a		
b		
m		
p		

link arrays

Figure 18.2 Arrays Corresponding to Initial Form of FP-tree: Root Node Only

18.2.3 Processing Transaction 1: *f*, *c*, *a*, *m*, *p*

Item *f* As this is the first item for the transaction we take the 'current node' to be the root node. In this case the current node does not have a descendant node with item name *f*, so a new node for item *f* is added numbered 1, with its parent node numbered 0 (indicating the root node) in Figure 18.4. Note that an item with name *f* and support count 1 is indicated by *f*/1 in Figure 18.3.

Adding a new node numbered *N*, for an item with name *Item* with its parent node numbered *P*

- A new node numbered *N* is added to the tree with item name *Item* and support count 1 as a descendant of the node numbered *P*.
- A new row, numbered *N*, is added to the *nodes* array with *itemname*, *count* and *parent* values *Item*, 1 and *P* respectively. The first unused *child* value for node *P* is set to *N*.
- The value of the row with index *Item* in both array *startlink* and array *endlink* is set to *N*.

Item *c*
The current node is now node 1, which does not have a descendant node with item name *c*, so a new node is added numbered 2, for item *c* with its parent node numbered 1.
Item *a*
The current node is now node 2, which does not have a descendant node with item name *a*, so a new node is added numbered 3, for item *a* with its parent node numbered 2.
Item *m*
The current node is now node 3, which does not have a descendant node with item name *m*, so a new node is added numbered 4, for item *m* with its parent node numbered 3.
Item *p*
The current node is now node 4, which does not have a descendant node with item name *p*, so a new node is added numbered 5, for item *p* with its parent node numbered 4.

This gives the partial tree and corresponding tables shown below.

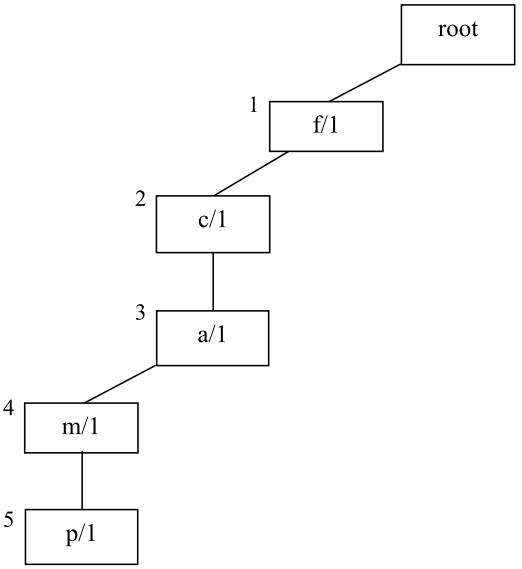

Figure 18.3 FP-tree After Processing Transaction 1

index	*item name*	*count*	*linkto*	*parent*
0	*root*			
1	*f*	1		0
2	*c*	1		1
3	*a*	1		2
4	*m*	1		3
5	*p*	1		4

nodes array

child1	*child2*
1	
2	
3	
4	
5	

child array

index	*startlink*	*endlink*
f	1	1
c	2	2
a	3	3
b		
m	4	4
p	5	5

link arrays

Figure 18.4 Arrays Corresponding to FP-tree After Processing Transaction 1

18.2.4 Processing Transaction 2: *f*, *c*, *a*, *b*, *m*

Items f, c and a
There is already a chain of nodes from the root to f, c, and a nodes in turn, so no changes are needed except to increase the counts of nodes 1, 2 and 3 and the corresponding rows of array *nodes* by one, giving Figures 18.5 and 18.6.

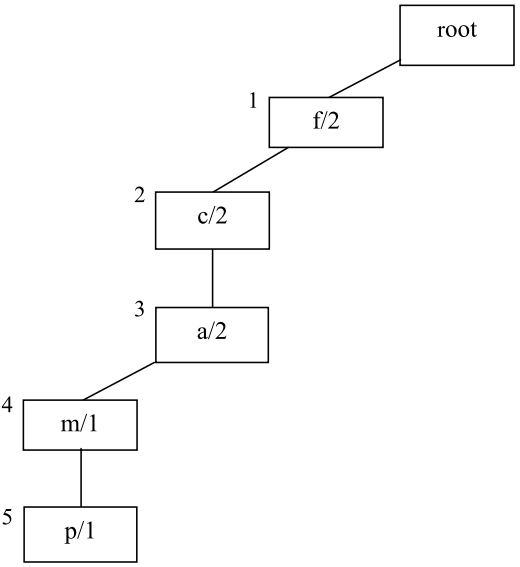

Figure 18.5 FP-tree After Processing First Three Items of Transaction 2

index	item name	count	linkto	parent
0	root			
1	f	2		0
2	c	2		1
3	a	2		2
4	m	1		3
5	p	1		4

nodes array

child1	child2
1	
2	
3	
4	
5	

child array

index	startlink	endlink
f	1	1
c	2	2
a	3	3
b		
m	4	4
p	5	5

link arrays

Figure 18.6 Arrays Corresponding to FP-tree After Processing First Three Items of Transaction 2

Item *b*

There is no descendant of the current node (the last node accessed), i.e. node 3, that has item name *b*, so a new node numbered 6 is added for item *b* with its parent node numbered 3 (Figures 18.7 and 18.8).

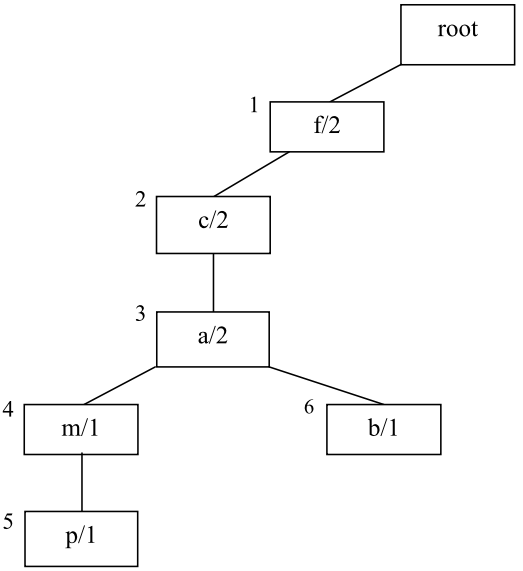

Figure 18.7 FP-tree After Processing First Four Items of Transaction 2

index	*item name*	*count*	*linkto*	*parent*
0	*root*			
1	*f*	2		0
2	*c*	2		1
3	*a*	2		2
4	*m*	1		3
5	*p*	1		4
6	*b*	1		3

nodes array

child1	*child2*
1	
2	
3	
4	6
5	

child array

index	*startlink*	*endlink*
f	1	1
c	2	2
a	3	3
b	6	6
m	4	4
p	5	5

link arrays

Figure 18.8 Arrays Corresponding to FP-tree After Processing First Four Items of Transaction 2

Item m

A new node numbered 7 is added for item m with its parent node numbered 6.

For the first time in this example the *endlink* array has a non-null value for a newly added node, as *endlink[m]* is 4. Because of this, a dashed line link is made from node 4 to node 7 for item m (Figures 18.9 and 18.10).

Making a 'dashed line' link for item *Item* across the tree from node A to node B

The *linkto* value in row A of the *nodes* array and the value of *endlink[Item]* are both set to B.

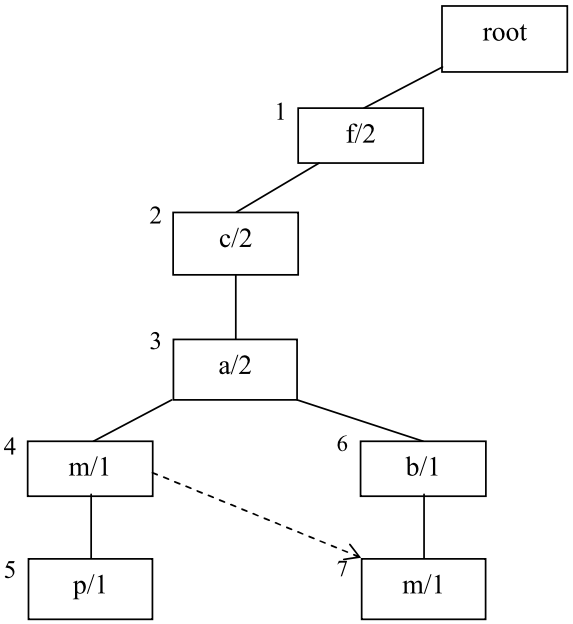

Figure 18.9 FP-tree After Processing All of Transaction 2

index	*item name*	*count*	*linkto*	*parent*
0	*root*			
1	*f*	2		0
2	*c*	2		1
3	*a*	2		2
4	*m*	1	7	3
5	*p*	1		4
6	*b*	1		3
7	*m*	1		6

nodes array

child1	*child2*
1	
2	
3	
4	6
5	
7	

child array

index	*startlink*	*endlink*
f	1	1
c	2	2
a	3	3
b	6	6
m	4	7
p	5	5

link arrays

Figure 18.10 Arrays Corresponding to FP-tree After Processing All of Transaction 2

18.2.5 Processing Transaction 3: *f*, *b*

Item *f*
The count value for node 1 in the tree and row 1 in the *nodes* array are both increased by 1.
Item *b*
There is no descendant of the current node, node 1, with item name *b* so a new node numbered 8 is added for item *b* with its parent node numbered 1.

The *endlink* array has a non-null value for the new node, as *endlink[b]* is 6. A dashed line link is made from node 6 to node 8 for item *b* (Figures 18.11 and 18.12).

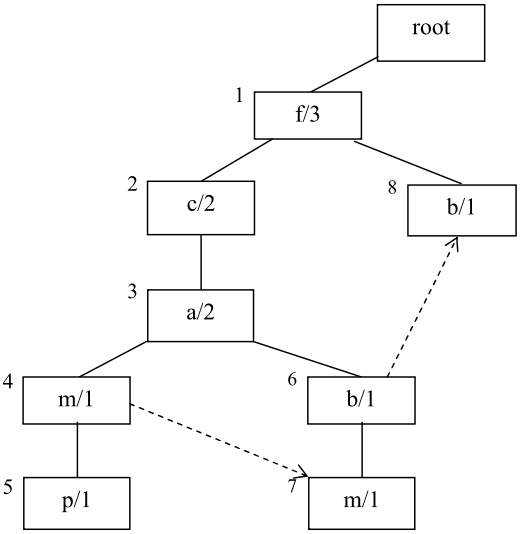

Figure 18.11 FP-tree After Processing All of Transaction 3

index	*item name*	*count*	*linkto*	*parent*
0	*root*			
1	f	3		0
2	c	2		1
3	a	2		2
4	m	1	7	3
5	p	1		4
6	b	1	8	3
7	m	1		6
8	b	1		1

nodes array

child1	*child2*
1	
2	8
3	
4	6
5	
7	

child array

index	*startlink*	*endlink*
f	1	1
c	2	2
a	3	3
b	6	8
m	4	7
p	5	5

link arrays

Figure 18.12 Arrays Corresponding to FP-tree After Processing All of Transaction 3

18.2.6 Processing Transaction 4: *c*, *b*, *p*

Item *c*
The current node (the root node) does not have a descendant node with item name *c*, so a new node is added numbered 9, for item *c* with its parent node numbered 0 (indicating the root node). A dashed line link is made from node 2 to node 9.
Item *b*
The current node is now node 9, which does not have a descendant node with item name *b*, so a new node is added numbered 10, for item *b* with its parent node numbered 9. A dashed line link is made from node 8 to node 10.
Item *p*
The current node is now node 10, which does not have a descendant node with item name *p*, so a new node is added numbered 11, for item *p* with its parent node numbered 10. A dashed line link is made from node 5 to node 11 (Figures 18.13 and 18.14).

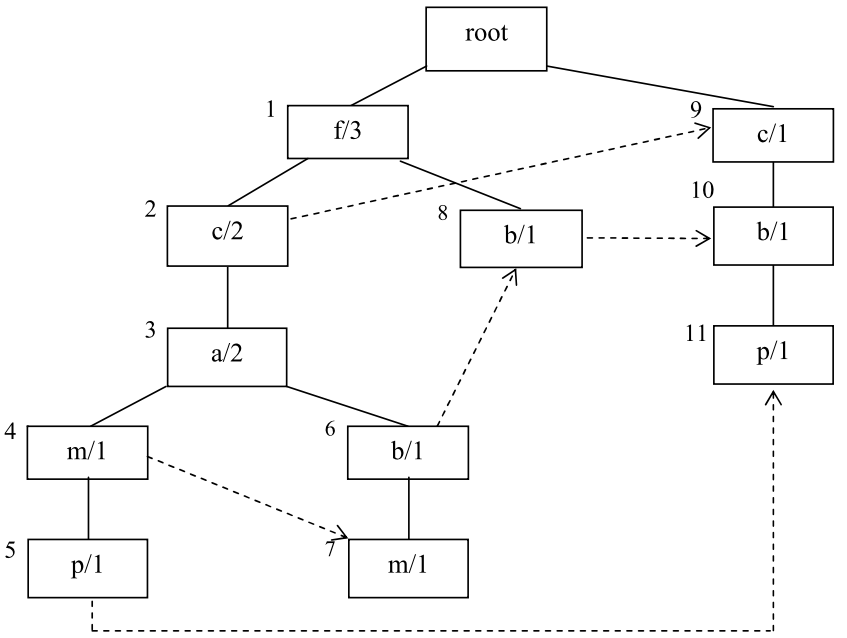

Figure 18.13 FP-tree After Processing Transaction 4

index	*item name*	*count*	*linkto*	*parent*
0	*root*			
1	*f*	3		0
2	*c*	2	9	1
3	*a*	2		2
4	*m*	1	7	3
5	*p*	1	11	4
6	*b*	1	8	3
7	*m*	1		6
8	*b*	1	10	1
9	*c*	1		0
10	*b*	1		9
11	*p*	1		10

nodes array

child1	*child2*
1	9
2	8
3	
4	6
5	
7	
10	
11	

child array

index	*startlink*	*endlink*
f	1	1
c	2	9
a	3	3
b	6	10
m	4	7
p	5	11

link arrays

Figure 18.14 Arrays Corresponding to FP-tree After Processing Transaction 4

18.2.7 Processing Transaction 5: *f*, *c*, *a*, *m*, *p*

There is already a chain of nodes from the root to *f*, *c*, *a*, *m* and *p* in turn, so no changes are needed except to increase the counts of nodes 1, 2, 3, 4 and 5 and the corresponding rows of array *nodes* by one. This gives the final FP-tree and corresponding set of arrays as follows (Figures 18.15 and 18.16).

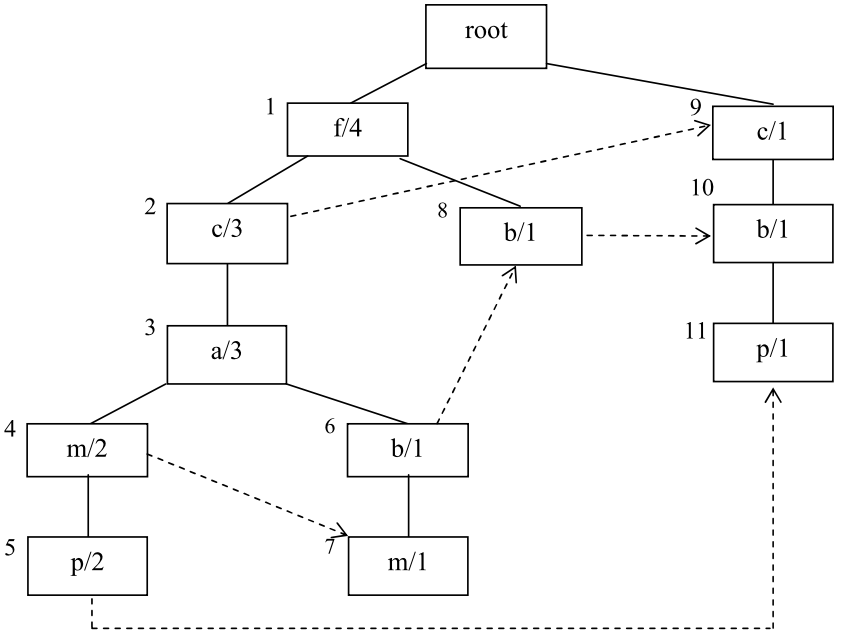

Figure 18.15 Final FP-tree After Processing Transaction 5

index	*item name*	*count*	*linkto*	*parent*
0	*root*			
1	*f*	4		0
2	*c*	3	9	1
3	*a*	3		2
4	*m*	2	7	3
5	*p*	2	11	4
6	*b*	1	8	3
7	*m*	1		6
8	*b*	1	10	1
9	*c*	1		0
10	*b*	1		9
11	*p*	1		10

nodes array

child1	*child2*
1	9
2	8
3	
4	6
5	
7	
10	
11	

child array

index	*startlink*	*endlink*
f	1	1
c	2	9
a	3	3
b	6	10
m	4	7
p	5	11

link arrays

Figure 18.16 Arrays Corresponding to Final FP-tree After Processing Transaction 5

Once the FP-tree has been created arrays *child* and *endlink* can be discarded. The contents of the tree are fully represented by arrays *nodes* and *startlink*.

18.3 Finding the Frequent Itemsets from the FP-tree

Having constructed the FP-tree, which is shown diagrammatically in Figure 18.15 and is represented by the arrays *nodes* and *startlink* shown in Figure 18.16, we can now analyse it to extract all the frequent itemsets for the transaction database.

We will illustrate the process by a series of diagrams and describe how the frequent itemset extraction process can be implemented in a recursive fashion by constructing a number of tables that are equivalent to reduced versions of the FP-tree.

We start by observing some general points.

- The dashed lines (links) in Figure 18.15 are not part of the tree itself (if there were links across the tree it would no longer be a tree structure). Rather, they are a way of keeping track of all the nodes with a particular name, e.g. b, wherever they occur in the tree. This will be very useful in what follows.
- The items used to label the nodes in each branch of the tree from the root downwards are always in the same order as the items in the *orderedItems* array, i.e. f, c, a, b, m, p. This is descending order of the support counts of the corresponding itemsets (e.g. $\{f\}$) in the transaction database, or equivalently the order of the items in the *orderedItems* array, which is repeated as Figure 18.17. (Not every branch of the tree includes all six of the items.)
- Although the nodes in Figure 18.15 are labelled with the names c, m, p etc. these are just the rightmost items in the itemsets to which the nodes correspond. Thus nodes 1, 2, 3, 4 and 5 correspond to the itemsets $\{f\}$, $\{f, c\}$, $\{f, c, a\}$, $\{f, c, a, m\}$ and $\{f, c, a, m, p\}$ respectively.

The *orderedItems* array is repeated here for convenience as Figure 18.17.

index	*orderedItems*
0	f
1	c
2	a
3	b
4	m
5	p

Figure 18.17 *orderedItems* array

The process of extracting all the frequent itemsets from the FP-tree is essentially a recursive one which can be represented by a call to a recursively-defined function *findFrequent* that takes four arguments:

- Two arrays representing the tree. Initially these are arrays *nodes* and *startlink*, corresponding to the original FP-tree. For future calls to the function these will be replaced by arrays *nodes2* and *startlink2* corresponding to a conditional FP-tree, as will be explained subsequently.
- Integer variable *lastitem*, which initially is set to the number of elements in the *orderedItems* array (6 in this example).
- A set named *originalItemset*, which is initially empty, i.e. {}.

We will start with an 'original itemset' with no members, i.e. {} and generate all possible one-item itemsets derived from it by adding a new item to its leftmost position in ascending order of the elements of *orderedItems*, i.e. $\{p\}$, $\{m\}$, $\{b\}$, $\{a\}$, $\{c\}$ and $\{f\}$ in that order[1]. For each of the itemsets that is frequent[2], say $\{m\}$, we next examine itemsets with an additional item in the leftmost position, e.g. $\{b, m\}$, $\{a, m\}$ or $\{c, m\}$ to find any that are frequent. Note that the additional item must be above m in the *orderedItems* array to preserve the conventional ordering of the items in an itemset. If we find a frequent itemset, e.g. $\{a, m\}$, we next construct itemsets with a further item in the leftmost position, e.g. $\{c, a, m\}$, check whether each one is frequent and so on. The effect is that having found a single-item itemset that is frequent we will go on to find all the frequent itemsets that end in the corresponding item before examining the next single-item itemset.

Constructing new itemsets by adding one new item at a time to the left, maintaining the same order as in the *orderedItems* array, is a very efficient way of proceeding. Having established that say $\{c, a\}$ is frequent, the only other itemset that needs checking is $\{f, c, a\}$ as f is the only item above c in *orderedItems*. It may be true (and it is true in this case) that some other itemset such as $\{c, a, m\}$ is also frequent but that will already have been dealt with at another stage.

Examining itemsets in this order also takes advantage of the downward closure property of itemsets. If we find that an itemset, say $\{b, m\}$ is infrequent

[1] This rather convoluted way of describing the generation of the itemsets {p}, $\{m\}$, $\{b\}$, $\{a\}$, $\{c\}$ and $\{f\}$ is for consistency with the description of the generation of two-item, three-item etc. itemsets that follows.

[2] All the single item itemsets must inevitably be frequent, as the items in the initial tree were selected from those in the transaction database on that basis. However this will often not be the case as we go on to use *findFrequent* recursively to analyse reduced versions of the FP-tree.

there is no point in examining any other itemsets with further items added. If any of them, say $\{f, c, b, m\}$ were frequent then by the downward closure property $\{b, m\}$ must be too, but we already know that it is not.

This strategy for generating frequent itemsets can be implemented in function *findFrequent* by a loop for variable *thisrow* through values from *lastitem*-1 down to zero.

- We set variable *nextitem* to *orderedItems[thisrow]* and then set *firstlink* to *startlink[nextitem]*.
- If *firstlink* is null we go on to the next value of *thisrow*.
- Otherwise we set variable *thisItemset* to be an expanded version of *originalItemset* with item *nextitem* as its leftmost item and then call function *condfptree* which takes four arguments: *nodes*, *firstlink*, *thisrow* and *thisItemset*.
- Function *condfptree* first sets variable *lastitem* to the value of *thisrow*. It then checks whether *thisItemset* is frequent. If it is, it goes on to generate a *conditional FP-tree* for that itemset in the form of arrays *nodes2* and *startlink2* and then calls *findFrequent* recursively with the two replacement arrays, together with *lastitem* and *thisItemset*, as arguments.

18.3.1 Itemsets Ending with Item *p*

Itemset {*p*} – expanded from original itemset {}

We start by establishing whether itemset $\{p\}$ is frequent. We can determine this from the FP-tree by examining the two linked p nodes (nodes 5 and 11) with support counts 2 and 1 respectively. The total count is 3, which is greater than or equal to the value of *minsupportcount* (i.e. 3 for this example). So **itemset {*p*} is frequent**.

It is straightforward to find the chain of p nodes from arrays *nodes* and *startlink* in the FP-tree (Figure 18.16). The value of *startlink[p]* is 5, the value in the *linkto* column of row 5 of the *nodes* array is 11 and the value in the *linkto* column of row 11 of the *nodes* array is null, indicating 'no further nodes'. Thus there is a chain of p nodes from node 5 to node 11.

Generating a conditional FP-tree for itemset $\{p\}$

Rather than going on, at this stage, to examine the frequency of other single-item itemsets $\{m\}$, $\{b\}$, $\{a\}$, $\{c\}$ and $\{f\}$, the algorithm first generates a sequence of two-item itemsets by extending the itemset $\{p\}$ by adding an item in the leftmost position. It does this for all the items that are above p in the

orderedItems array in turn. Thus the two-item itemsets $\{m, p\}$, $\{b, p\}$, $\{a, p\}$, $\{c, p\}$ and $\{f, p\}$ are examined in turn. If any of them is frequent its conditional FP-tree is constructed and a sequence of three-item itemsets is generated by extending the two-item itemset by adding an item in the leftmost position. The process continues in this fashion until the whole tree structure has been examined. At each stage when the current itemset is expanded by adding an extra item in the leftmost position, only those items in the *orderedItems* array (Figure 18.17) above the one previously in the leftmost position are considered.

We now need to check whether any two-item itemsets formed by adding an additional item to itemset $\{p\}$ are also frequent. To do this we first construct a *conditional FP-tree* for itemset $\{p\}$. This is a reduced version of the original FP-tree that contains only the branches that begin at the root and end at the two nodes labelled p, but with the nodes renumbered and often with different support counts. (It may be helpful to look ahead to Figures 18.20 and 18.21 at this point.)

Initialisation

Diagrammatically we can represent the initial state of the FP-tree by a single unnumbered node, representing the root.

We will represent the evolving tree by the contents of four arrays, all initially empty:

- A two-dimensional array *nodes2*, with a numerical index that will correspond to the numbering of the nodes in the tree. The names given to the columns of this array are the same as those for array *nodes* in Section 18.2.
- A single-dimensional array *oldindex*, which for each node holds the number of the corresponding node in the tree from which the evolving conditional FP-tree is derived (initially the FP-tree shown in Figure 18.15).
- Single-dimensional arrays *startlink2* and *lastlink* indexed by the names of some or all of the itemsets in the *orderedItems* array.

We again work through the chain of linked p nodes, this time adding branches to an evolving conditional FP-tree for itemset $\{p\}$ and values to the four equivalent arrays as we do so.

First Branch

Add the five nodes in the leftmost branch of the FP-tree (Figure 18.15), numbering from the bottom upwards, as a branch leading up to the root, all with the support count of the lowest node (i.e. the one with *itemname* p).

Values corresponding to each node in turn are added to the four arrays, as described in the box below (note that this is not yet a complete description).

Adding a branch that ends in a node with support count *Count*
Version 1

For each node

1. Set variables *thisitem* and *thisparent* to the values of *itemname* and *parent* for the original node, respectively. Add a new row to the *nodes2* array, with the value of *itemname* set to *thisitem*. Set the value of *count* (for all the nodes) to *Count*.
2. Set the value in the *oldindex* array to the number of the node in the tree from which the evolving conditional FP-tree is being derived.
3. Set the values of *startlink2[thisitem]* and *lastlink[thisitem]* to the new row number.
4. If the value of *thisparent* is not zero or null, set the value of *parent* in the *nodes2* array to the number of the following row.

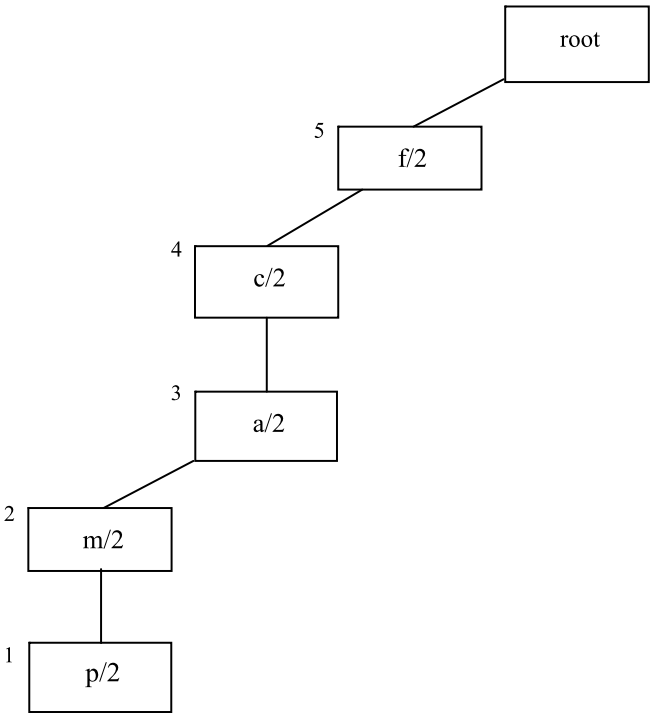

Figure 18.18 Conditional FP-tree for $\{p\}$ – first branch only

Note that in Figure 18.18 the numbering of the nodes is different from that in Figure 18.15. It reflects the order in which this new tree has been generated, working from bottom (the p node) to top (the root) for each branch. The root node has not been numbered and the other nodes are numbered from 1 onwards.

index	*item name*	*count*	*linkto*	*parent*
1	p	2		2
2	m	2		3
3	a	2		4
4	c	2		5
5	f	2		

nodes2 array

oldindex
5
4
3
2
1

oldindex

index	*startlink2*	*lastlink*
p	1	1
m	2	2
a	3	3
c	4	4
f	5	5

link arrays

Figure 18.19 Arrays Corresponding to Conditional FP-tree for $\{p\}$ – first branch only

The values in the *nodes2*, *oldindex*, *startlink2* and *lastlink* arrays corresponding to the first branch are shown in Figure 18.19.

The null value in the *parent* column of node 5 indicates a link to the root node. The use of the *linkto* column in array *nodes2* will be explained when we go on to add the second branch. The use of the array *oldindex* will be explained in Section 18.3.2.

Note that the support counts of the branch in Figure 18.18 are different from those of the corresponding branch in the FP-tree (Figure 18.15). When we constructed the original FP-tree we thought of a node such as node 3 as representing an itemset $\{f, c, a\}$ with support count 3. All the nodes in the branch from node 1 down to node 5 represented itemsets beginning with f, e.g. node 4 represented $\{f, c, a, m\}$. We need to think of a conditional FP-tree in a different way, working from the bottom of each branch to the top. The lowest node (now numbered 1) in Figure 18.18 now represents (part of) itemset $\{p\}$, node 2 represents itemset $\{m, p\}$, nodes 3, 4 and 5 represent itemsets $\{a, m, p\}$, $\{c, a, m, p\}$ and $\{f, c, a, m, p\}$ respectively. In all cases the itemset ends with

item p rather than starting with item f. Looking at Figure 18.18 this way, the support counts for the a, c and f nodes cannot be 3, 3 and 4 respectively as they were in the FP-tree. If there are two transactions that include item p there cannot be more than 2 transactions that include items a and p together, or any other such combination.

For this reason the best approach to constructing the conditional FP-tree for $\{p\}$ is to construct the tree bottom-up, branch by branch, using the counts of the p nodes. Each new node entered in the tree 'inherits' the support count of the p node at the bottom of the branch.

Second Branch

We now add the second and final branch that ends in a node with *itemname* p in the FP-tree.

This gives the final version of the conditional FP-tree for itemset $\{p\}$ shown in Figure 18.20.

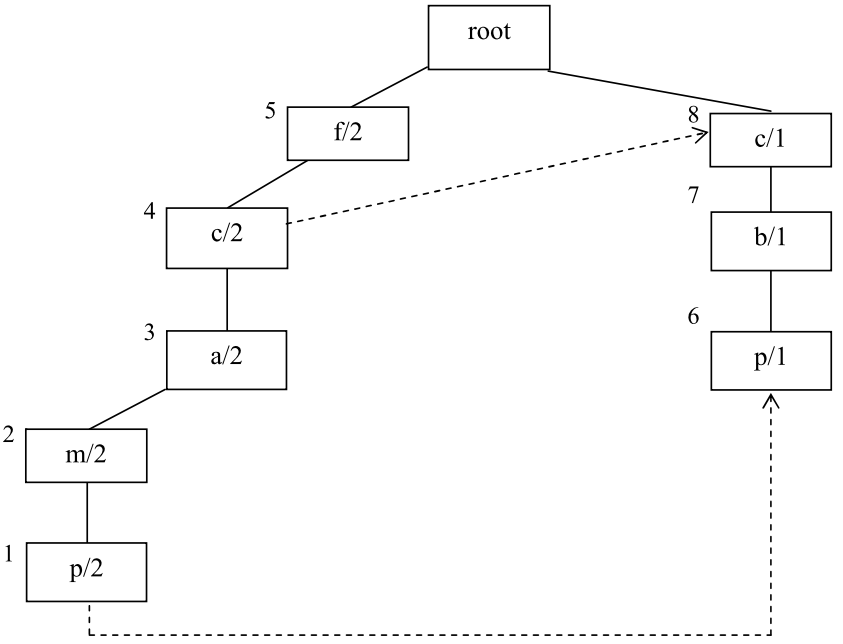

Figure 18.20 Conditional FP-tree for $\{p\}$ – final version

The important difference from adding the first branch is that now the dashed line links have been added for nodes p and c. These are essential for determining whether itemsets are frequent at each stage of the extraction process.

The algorithm for adding additional nodes needs to be augmented to deal with this. For example after node 6 (a second p node) is added, we can tell that there is already a p node in the tree by the non-null value in *lastlink[p]*. The current value of *lastlink[p]* is 1 so we set both the *linkto* value for row 1 and the new value of *lastlink[p]* to the current row number (i.e. 6). This effectively creates a chain of two p nodes from node 1 to node 6. A similar procedure occurs when node 8 (a c node) is added.

A revised version of the algorithm for adding a new branch is given in the box below (but this is still not a complete description).

Adding a branch that ends in a node with support count *Count*
Version 2

For each node

1. Set variables *thisitem* and *thisparent* to the values of *itemname* and *parent* for the original node, respectively. Add a new row to the *nodes2* array, with the value of *itemname* set to *thisitem*. Set the value of *count* (for all the nodes) to *Count*.
2. Set the value in the *oldindex* array to the number of the node in the tree from which the evolving conditional FP-tree is being derived.
3. Set *lastval* to *lastlink[thisitem]*.

   IF *lastval* is not null, set both the *linkto* value in row *lastval* and *lastlink[thisitem]* to the current row number.

   ELSE set the values of *startlink2[thisitem]* and *lastlink[thisitem]* to the current row number.
4. If the value of *thisparent* is not zero or null, set the value of *parent* in the *nodes2* array to the number of the following row.

The values in the *nodes2*, *oldindex*, *startlink2* and *lastlink* arrays corresponding to the final version of the conditional FP-tree for itemset $\{p\}$ are shown in Figure 18.21.

index	*item name*	*count*	*linkto*	*parent*
1	*p*	2	6	2
2	*m*	2		3
3	*a*	2		4
4	*c*	2	8	5
5	*f*	2		
6	*p*	1		7
7	*b*	1		8
8	*c*	1		

nodes2 array

oldindex
5
4
3
2
1
11
10
9

oldindex

index	*startlink2*	*lastlink*
p	1	6
m	2	2
a	3	3
c	4	8
f	5	5
b	7	7

link arrays

Figure 18.21 Arrays Corresponding to Conditional FP-tree for $\{p\}$ – final version

The null values in the *parent* column of nodes 5 and 8 indicate links to the root node. The non-null values in the *linkto* column of array *nodes2* correspond to 'dashed line' links between nodes across the tree.

Two-item Itemsets

Having constructed the conditional FP-tree for itemset $\{p\}$, there are five two-item itemsets to examine, starting with $\{m, p\}$. In each case we do it by extracting the part of the tree that contains only the branches that begin at the root and end at each of the nodes labelled m (or similarly for each of the other items b, a, c, and f in turn). Note that the nodes in the conditional FP-tree are numbered sequentially from 1 (in the order they are generated) each time.

To implement the creation and examination of the two-item itemsets expanded from $\{p\}$ we make a recursive call from function *condfptree* to function *findFrequent* with four arguments: *nodes2*, *startlink2*, *lastitem* and *thisItemset*. The last of these has the value $\{p\}$.

A sequence of itemsets with two items is now generated from the conditional FP-tree for itemset $\{p\}$ by making a loop through the *orderedItems* array from

row *lastitem*-1 to row zero. As *lastitem* is now 5, this means that the items used as a new leftmost item for the expanded itemsets are m, b, a, c and f in that order (but not p).

Itemsets $\{m, p\}$, $\{b, p\}$, $\{a, p\}$ and $\{c, p\}$ – expanded from original itemset $\{p\}$

{m, p}: There is only one m node, which has a count of 2. So $\{m, p\}$ is infrequent (Figure 18.22).

{b, p}: There is only one b node, which has a count of 1. So $\{b, p\}$ is infrequent (Figure 18.23).

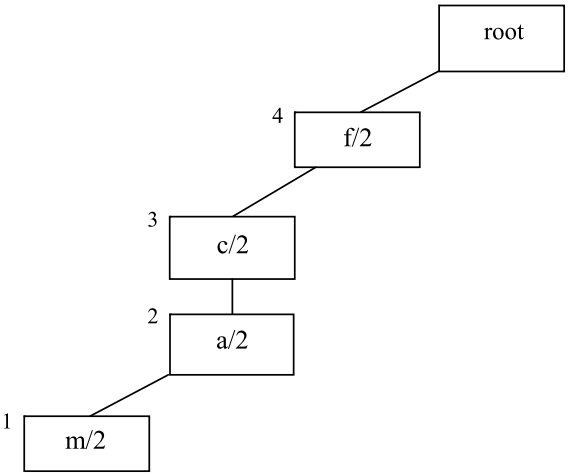

Figure 18.22 Conditional FP-tree for $\{m, p\}$

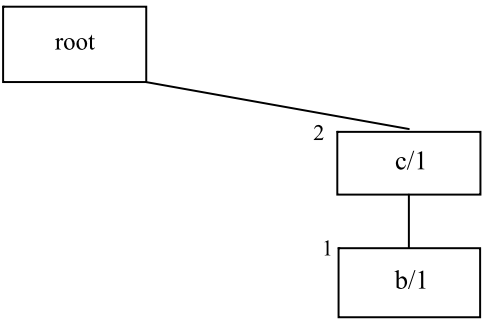

Figure 18.23 Conditional FP-tree for $\{b, p\}$

{a, p}: There is only one a node, which has a count of 2. So $\{a, p\}$ is infrequent (Figure 18.24).

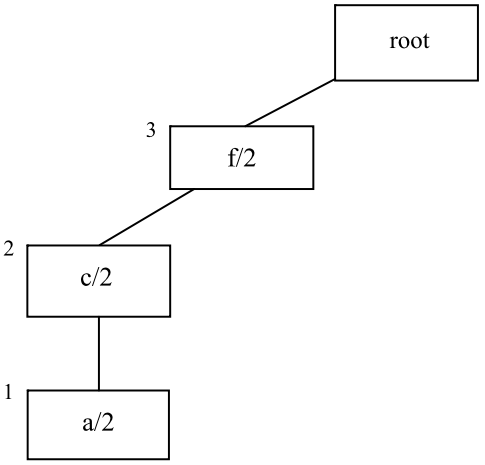

Figure 18.24 Conditional FP-tree for $\{a, p\}$

{c, p}: There are two c nodes, with a total count of 3. So **$\{c, p\}$ is frequent** (Figure 18.25).

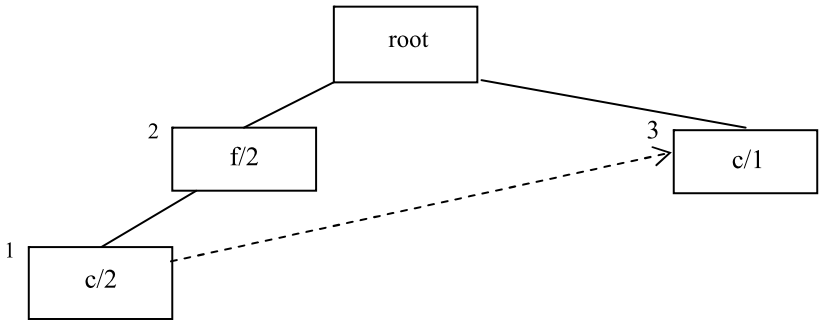

Figure 18.25 Conditional FP-tree for $\{c, p\}$

Before going on to examine $\{f, p\}$ we now generate all **three-item itemsets** formed by adding an additional item to the leftmost position of $\{c, p\}$. We only consider those items above c in the *orderedItems* array. There is only one, i.e. f. So we start by generating the conditional FP-tree for $\{f, c, p\}$.

We implement this by making a recursive call from function *condfptree* to function *findFrequent* with four arguments: the arrays *nodes2* and *startlink2* that correspond to Figure 18.25, *lastitem* (which is now 1) and *thisItemset*, which is now $\{c, p\}$.

Itemset $\{f, c, p\}$ – expanded from original itemset $\{c, p\}$
There is only one f node, which has a count of 2 (Figure 18.26). So $\{f, c, p\}$ is infrequent. We go back to examining the two-item itemsets, the next of which is $\{f, p\}$.

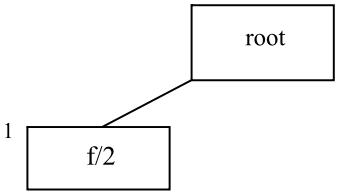

Figure 18.26 *Conditional FP-tree* for $\{f, c, p\}$

Itemset $\{f, p\}$ – expanded from original itemset $\{p\}$
There is only one f node, which has a count of 2. So $\{f, p\}$ is infrequent (Figure 18.27).

We have found two frequent itemsets ending with item p: $\{p\}$ and $\{c, p\}$. There cannot be any other frequent itemsets ending with p. For example if $\{f, c, b, p\}$ were frequent then by the downward closure property all its non-empty subsets would be frequent too. That would include itemset $\{b, p\}$, which we

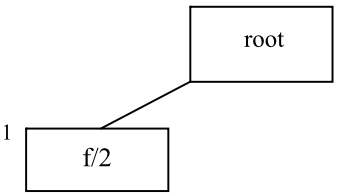

Figure 18.27 *Conditional FP-tree* for $\{f, p\}$

already know is infrequent. There are 32 possible itemsets with p as the rightmost item and in descending order of the items in the *orderedItems* array. We have only needed to examine seven of them (two frequent and five infrequent).

For space reasons we will not examine all the other single-item itemsets and those constructed by expanding them by adding additional items in the leftmost position. However we will examine itemset $\{m\}$ and its derivatives as this will illustrate some important additional points.

18.3.2 Itemsets Ending with Item *m*

Itemset {*m*} – expanded from original itemset {}
The conditional FP-tree for $\{m\}$ is shown as Figure 18.28.

Note that nodes 2, 3 and 4 inherit a support count of 2 from node 1 and a support count of 1 from node 5. For that reason their (total) support counts are shown as 3.

There are two m nodes, with a total count of 3. So **{*m*} is frequent**.

In constructing the tree bottom-up it is important to distinguish between the case that applies here, where the *parent* of node 6 is an a node (node 2) that has already been entered in the tree and the case where the parent is a different a node, not yet in the tree, which needs to be created.

Figure 18.29 shows the state of the four arrays as node 6 in Figure 18.28 is about to be added to the tree.

The first part of the processing is the same as for all other nodes. The new node is part of a branch that ends in a m node with support count 1. As it happens, the node was also numbered 6 in the original FP-tree, so variables *thisitem* and *thisparent* are taken from row 6 of the *nodes* array and set to b and 3 respectively. A new row, row 6, is added to *nodes2* with the values of *itemname* and *parent* set to b and 3 respectively. The value of element 6 in *oldindex* is set to 6. Next *lastval* is set to *lastlink[b]* which is null, so both *startlink2[b]* and *lastlink[b]* are set to 6.

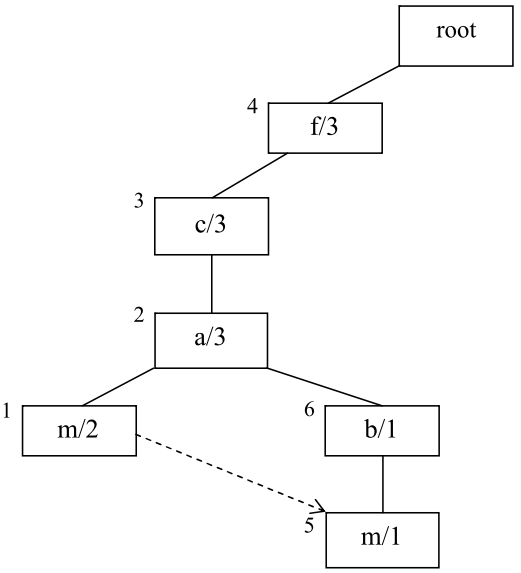

Figure 18.28 Conditional FP-tree for itemset $\{m\}$

index	*item name*	*count*	*linkto*	*parent*
1	m	2	5	2
2	a	2		3
3	c	2		4
4	f	2		
5	m	1		

nodes2 array

oldindex
4
3
2
1
7

oldindex

index	*startlink2*	*lastlink*
m	1	5
a	2	2
c	3	3
f	4	4

link arrays

Figure 18.29 Arrays corresponding to Conditional FP-tree for itemset $\{m\}$ – first five nodes only

It is at the final stage that the processing of this node differs from the algorithm used up to now. We check whether the value of *thisparent* (i.e. 3) is

already in the *oldindex* array. Unlike for all the examples shown previously, it is there in position 2, implying that the b node has a parent, node 2, which is already present in the evolving tree structure. This in turn implies that the new node 6 needs to be linked to the part of the tree structure that has already been created. There are three stages to this.

– The value of *parent* in row 6 of *nodes2* is set to 2.

– The adding of additional nodes for the current branch is aborted.

– The chain of parent nodes in the *nodes2* array is followed from row 2, up to immediately before the root, i.e. from 2 to 3 to 4, with the support count being increased by the support count of the node at the bottom of the branch (i.e. by 1) at each stage.

This concludes the construction of the arrays corresponding to the conditional FP-tree for itemset $\{m\}$, giving Figure 18.30.

index	*item name*	*count*	*linkto*	*parent*
1	m	2	5	2
2	a	3		3
3	c	3		4
4	f	3		
5	m	1		6
6	b	1		2

nodes2 array

oldindex
4
3
2
1
7
6

oldindex

index	*startlink2*	*lastlink*
m	1	5
a	2	2
c	3	3
f	4	4
b	6	6

link arrays

Figure 18.30 Arrays corresponding to Conditional FP-tree for itemset $\{m\}$ – all nodes

This leads to a revised and final version of the algorithm for adding a branch.

Adding a branch that ends in a node with support count *Count*
Final version

For each node

1. Set variables *thisitem* and *thisparent* to the values of *itemname* and *parent* for the original node, respectively. Add a new row to the *nodes2* array, with the value of *itemname* set to *thisitem*. Set the value of *count* (for all the nodes) to *Count*.
2. Set the value in the *oldindex* array to the number of the node in the tree from which the evolving conditional FP-tree is being derived.
3. Set *lastval* to *lastlink[thisitem]*.

   IF *lastval* is not null, set both the *linkto* value in row *lastval* and *lastlink[thisitem]* to the current row number.

   ELSE set the values of *startlink2[thisitem] and lastlink[thisitem]* to the current row number.
4. If the value of *thisparent* is not zero or null, test whether the value of *thisparent* is already in array *oldindex* at position *pos*.

   If it is {

   (a) Set the value of *parent* for the current row of *nodes2* to *pos*.

   (b) Abort the adding of additional nodes for the current branch.

   (c) Follow the chain of parent nodes in the *nodes2* array from row *pos* up to immediately before the root, increasing the support count by *Count* for each one.

   }

   Otherwise set the value of *parent* for the current row of *nodes2* to the number of the following row.

Having done this the algorithm now goes on to consider the four possible two-item itemsets $\{b, m\}$, $\{a, m\}$, $\{c, m\}$ and $\{f, m\}$ in turn (only items above m in the *orderedItems* array need to be considered for the leftmost position). The relevant conditional FP-trees in the order in which they are constructed are as follows.

Itemsets $\{b, m\}$ and $\{a, m\}$ – expanded from original itemset $\{m\}$

{b, m}: There is only one b node, which has a count of 1. So $\{b, m\}$ is infrequent. Note that the count of 1 has been inherited from node 1 by nodes 2, 3 and 4 (Figure 18.31).

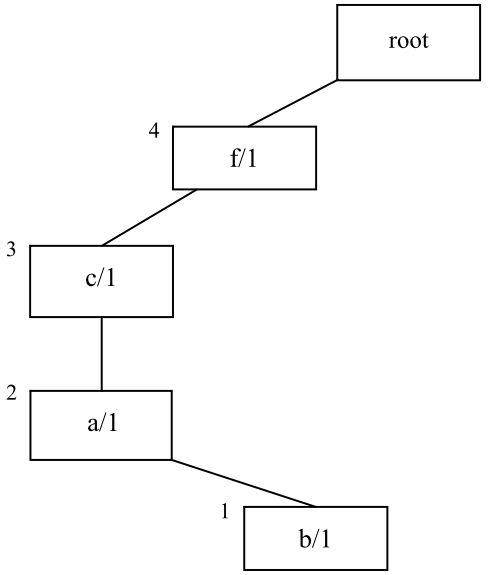

Figure 18.31 Conditional FP-tree for itemset $\{b, m\}$

{a, m}: There is only one a node, which has a count of 3. So **$\{a, m\}$ is frequent** (Figure 18.32).

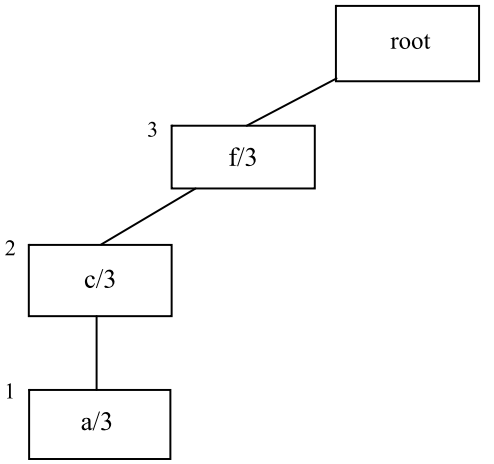

Figure 18.32 Conditional FP-tree for itemset $\{a, m\}$

We now examine all the three-item itemsets constructed by expanding $\{a,\ m\}$ by adding an item in the leftmost position. Only items above a in the *orderedItems* array need to be considered, i.e. c then f.

Itemset $\{c, a, m\}$ – expanded from original itemset $\{a, m\}$

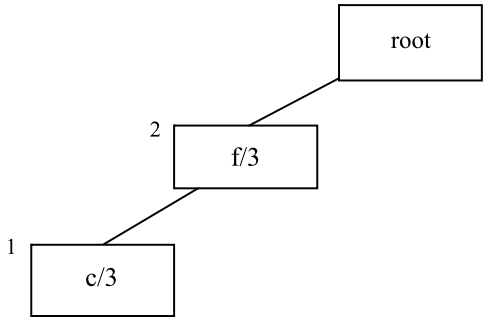

Figure 18.33 Conditional FP-tree for itemset $\{c, a, m\}$

There is only one c node, which has a count of 3 (Figure 18.33). So **$\{c,\ a,\ m\}$ is frequent**.

We now examine all the four-item itemsets constructed by expanding $\{c,\ a,\ m\}$ by adding an item in the leftmost position. Only items above c in the *orderedItems* array need to be considered, i.e. f.

Itemset $\{f, c, a, m\}$ – expanded from original itemset $\{c, a, m\}$

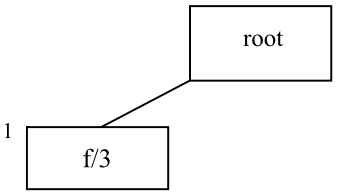

Figure 18.34 Conditional FP-tree for itemset $\{f, c, a, m\}$

There is only one f node, which has a count of 3 (Figure 18.34). So **$\{f, c, a, m\}$ is frequent**.

As there is no item above f in *orderedItems* and there are no other four-item itemsets expanded from $\{c, a, m\}$ to be considered, the examination of itemsets expanded from $\{c, a, m\}$ is concluded.

This can be implemented by adding a test to function *condfptree* so that having established that an itemset is frequent the function only goes on to generate the conditional FP-tree etc. if the value of *lastitem* is greater than zero.

Itemset $\{f, a, m\}$ – expanded from original itemset $\{a, m\}$

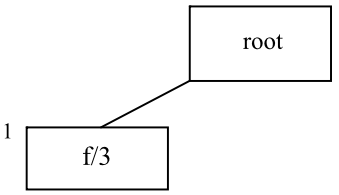

Figure 18.35 Conditional FP-tree for itemset $\{f, a, m\}$

There is only one c node, which has a count of 3 (Figure 18.35). So **$\{f, a, m\}$ is frequent**.

As there is no item above f in *orderedItems* the examination of itemsets with three items that are expanded versions of $\{a, m\}$ is concluded.

Itemset $\{c, m\}$ – expanded from original itemset $\{m\}$

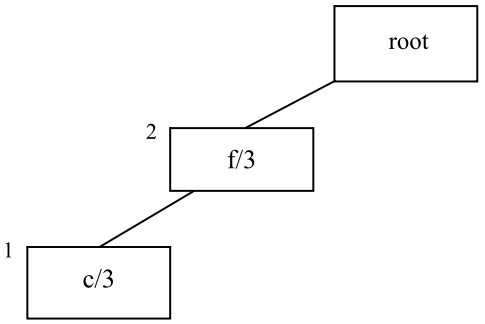

Figure 18.36 Conditional FP-tree for itemset $\{c, m\}$

There is only one c node, which has a count of 3 (Figure 18.36). So **$\{c, m\}$ is frequent**.

We now examine all the three-item itemsets constructed by expanding $\{c, m\}$ by adding an item in the leftmost position. Only items above c in the *orderedItems* array need to be considered, i.e. f.

Itemset $\{f, c, m\}$ – expanded from original itemset $\{c, m\}$

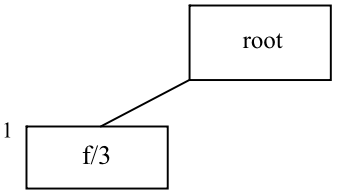

Figure 18.37 Conditional FP-tree for itemset $\{f, c, m\}$

There is only one f node, which has a count of 3 (Figure 18.37). So **$\{f, c, m\}$ is frequent**.

As there is no item above f in *orderedItems* the examination of itemsets with three items that are expanded versions of $\{c, m\}$ is concluded.

Itemset $\{f, m\}$ – expanded from original itemset $\{m\}$

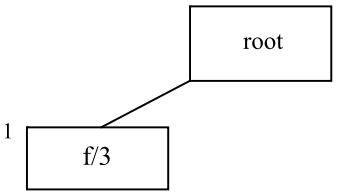

Figure 18.38 *Conditional FP-tree* for itemset $\{f, m\}$

There is only one f node, which has a count of 3 (Figure 18.38). So **$\{f, m\}$ is frequent**.

As there is no item above f in *orderedItems* and there are no more two-item itemsets to be considered, the examination of itemsets with final item m is concluded.

This time we have found 8 frequent itemsets ending with item m (there cannot be any others) and have examined only one infrequent itemset. There are 16 possible itemsets with m as the rightmost item that are in descending order of the items in the *orderedItems* array. We have only needed to examine a total of nine of them.

18.4 Chapter Summary

This chapter introduces the *FP-growth* algorithm for extracting frequent itemsets from a database of transactions. First the database is processed to produce a data structure called a *FP-tree*, then the tree is processed recursively by

constructing a sequence of reduced trees known as *conditional FP-trees*, from which the frequent itemsets are extracted. The algorithm has the very desirable feature of requiring only two scans through the database.

18.5 Self-assessment Exercises for Chapter 18

1. Draw the conditional FP-tree for itemset $\{c\}$.
2. How can the support count for $\{c\}$ be determined from the conditional FP-tree? What is it?
3. Is itemset $\{c\}$ frequent?
4. What are the contents of the four arrays corresponding to the conditional FP-tree for itemset $\{c\}$?

Reference

[1] Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. *SIGMOD Record*, *29*(2), 1–12. Proceedings of the 2000 ACM SIGMOD international conference on management of data, ACM Press.