CHAPTER IV

MAXIMAL K-FREQUENT INTERVALS

4.1 INTRODUCTION

As discussed earlier in Chapter I, the notion of maximal k-frequent intervals was introduced by Lin [Lin03] and it has the potential to contribute to knowledge discovery in many real-life application areas. In this chapter, the contributions made in the present work to the problem of mining maximal k-frequent intervals are discussed. In Section 4.2, some important definitions relevant to the current and subsequent chapters are introduced. A precise definition of the problems discussed in this chapter is presented in Section 4.3. In Section 4.4, Lin's method [Lin03] for mining maximal k-frequent intervals is reviewed. To the best of our knowledge, this is the only method that existed earlier for mining maximal k-frequent intervals. To identify maximal k-frequent intervals, Lin had introduced a data-structure called I-Tree. In Section 4.5, a more time-efficient construction of the I-Tree structure is presented and this leads to an improvement in Lin's method [Lin03] for mining maximal k-frequent intervals. In Section 4.6, an asymptotically faster method with new data-structures is proposed for determining the set of maximal k-frequent intervals with respect to an interval transaction database. In Section 4.7, a method is presented that efficiently uses the set of maximal k-frequent intervals to determine whether any given non-empty interval is k-frequent.
with respect to an interval transaction database. In Section 4.8, experimental results are reported. Section 4.9 summarizes the contributions made in this chapter.

4.2 SUPPORT, K-FREQUENT INTERVALS AND MAXIMAL K-FREQUENT INTERVALS

In the following definitions, TDB is an interval transaction database.

DEFINITION 4.1 An interval transaction T supports a non-empty interval I or alternatively, a non-empty interval I is supported by T if $I \subseteq \text{intv}(T)$.

DEFINITION 4.2 A set of interval transactions $S$ supports a non-empty interval I or alternatively, a non-empty interval I is supported by $S$ if I is supported by every interval transaction in $S$.

DEFINITION 4.3 Absolute support. The absolute support of a non-empty interval I with respect to TDB, denoted by $\sigma(I, \text{TDB})$, is defined to be the number of interval transactions in TDB that support I.

DEFINITION 4.4 k-frequent interval. For $k > 0$, a non-empty interval I is defined to be k-frequent with respect to TDB if $\sigma(I, \text{TDB}) \geq k$. The set of k-frequent intervals with respect to TDB is denoted by $F(k, \text{TDB})$.

DEFINITION 4.5 Maximal k-frequent interval. For $k > 0$, a non-empty interval I is defined to be a maximal k-frequent interval with respect to TDB if $I \in F(k, \text{TDB})$ and there is no other interval J such that $I \subseteq J$ and $J \in F(k, \text{TDB})$. The set of maximal k-frequent intervals with respect to TDB is denoted by $M(k, \text{TDB})$. 
EXAMPLE 4.1 Consider the following interval transaction database TDB (Table 4.1) containing ten interval transactions:

Table 4.1 An interval transaction database

<table>
<thead>
<tr>
<th>Tid</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>[1,2]</td>
<td>[2,6]</td>
<td>[3,5]</td>
<td>[3,9]</td>
<td>[4,5]</td>
<td>[8,10]</td>
<td>[8,9]</td>
<td>[8,9]</td>
<td>[9,10]</td>
<td></td>
</tr>
</tbody>
</table>

Here, $\sigma([4, 7], \text{TDB}) = 1$; $[4, 7] \notin F(3, \text{TDB})$; $[8, 9] \in M(3, \text{TDB})$; $[8, 9] \in M(2, \text{TDB})$; $[8, 9] \notin M(1, \text{TDB})$

4.3 PROBLEM DEFINITION

Two problems are addressed in this chapter—

(i) Given an interval transaction database $S$ and a minimum absolute support $k$, it is required to determine $M(k, S)$.

(ii) Given an interval transaction database $S$ and a non-empty interval $I$, it is required to determine whether $I \in F(k, S)$.

4.4 PREVIOUS WORK

In [Lin03], Lin proposed a two-stage method for determining maximal $k$-frequent intervals. To the best of our knowledge, this is the only method that existed earlier for mining maximal $k$-frequent intervals. In Section 4.4.1, a brief description of Lin's method for determining maximal $k$-frequent intervals is
4.4.1 Lin's Method

Please note: In the present work, an endpoint is defined as an ordered pair \((t, b)\) where \(t\) is a domain element and \(b\) is in \([\text{'}\text{'}, \text{'}\text{'})\) (Chapter III, Definition 3), whereas Lin considers a domain element itself as an endpoint. In sections 4.4 and 4.5, Lin’s definition of endpoints will be used.

Let \(LL = \{l_1, l_2, \ldots, l_{|LL|}\}\) be the set of distinct left endpoints in a database of intervals with \(l_1 < l_2 < \ldots < l_{|LL|}\). \(|RR|\) is the number of distinct right endpoints and \(r_{\text{max}}\) denotes the largest right endpoint in the database. Lin introduces a data-structure called I-Tree to determine maximal k-frequent intervals.

![Figure 4.1 The I-Tree structure](image-url)
Description of the I-Tree structure: Each node in the I-Tree represents an interval. An I-Tree contains two lists - a header list H and a non-header list for every node in H. The nodes in H represent the intervals \([l_1, r_{\text{max}}], [l_2, r_{\text{max}}], \ldots, [l_n, r_{\text{max}}]\) and they are linked together in ascending order of their left endpoints. A node in H representing the interval \([l_i, r_{\text{max}}]\) is also the head of a non-header list denoted by \(L_i\). All the nodes in the non-header list \(L_i\) represent the intervals with the same left endpoint \(l_i\) and are linked together in descending order of their right endpoints. Each node \(q\), except the head, in a non-header list contains the fields - \(q_{\text{left}}, q_{\text{right}}, q_{\text{count}}, q_{\text{next-node}}, \) where \([q_{\text{left}}, q_{\text{right}}]\) is the interval represented by the node \(q\), \(q_{\text{count}}\) is the number of occurrences of the interval \([q_{\text{left}}, q_{\text{right}}]\) in the given database of intervals and \(q_{\text{next-node}}\) is the address of the node succeeding node \(q\) in the non-header list. On the other hand, each node \(p\) in the header list H contains the fields - \(p_{\text{left}}, p_{\text{right}}, p_{\text{count}}, p_{\text{next-head}}, p_{\text{next-node}}, \) where \(p_{\text{next-head}}\) is the address of the node succeeding node \(p\) in the header list H. The rest of the fields are the same as those of the nodes in a non-header list.

Lin's method for determining maximal k-frequent intervals has two stages—

Stage 1: Construction of the I-Tree. For each interval in the given database of intervals, the point of insertion in the I-Tree is located after an initial sequential search of the header list H, followed by a sequential search of some non-header list \(L_i\), if necessary. Depending on the search result, either a new node is inserted (in H and/or \(L_i\)) or the count field of the appropriate node is incremented.
Stage 2: The Preorder Traversal (PT) algorithm. In this stage, the nodes in the non-header lists $L_1, L_2, \ldots, L_{|L|}$ are processed in sequence. While processing a node $q$ in $L_i$, the interval count of the previous node is added to $q_{\text{count}}$. If $q_{\text{count}}$ now becomes greater than or equal to $k$, then $q$ is a maximal $k$-frequent node.

If $q$ is a maximal $k$-frequent node: The processing of rest of the nodes in $L_i$ is abandoned and $q_{\text{count}}$ is remembered in a variable $r_{\text{prune}}$. Later, when the nodes in $L_j$ with $j > i$ are processed, the nodes with right endpoints less than or equal to $r_{\text{prune}}$ need not be processed. If another maximal $k$-frequent node is discovered while processing $L_j$ ($j > i$), the value of $r_{\text{prune}}$ is updated accordingly.

If $q$ is not a maximal $k$-frequent node: If a node $j$ is found in $L_{i+1}$ (provided $L_{i+1}$ exists) such that $j_{\text{right}} = q_{\text{right}}$, then $q_{\text{count}}$ is added to $j_{\text{count}}$. If however such a node $j$ does not exist in $L_{i+1}$, then a node $j$ is created with $j_{\text{left}} = l_{i+1}, j_{\text{right}} = q_{\text{right}}, j_{\text{count}} = q_{\text{count}}$ and is inserted at the appropriate place in $L_{i+1}$.

If $n$ is the total number of intervals in the given database, then in the first stage, the number of nodes in the I-Tree is $O(n)$. However, many more nodes may be processed by the PT algorithm because of the insertion of new nodes into the I-Tree in the second stage. The new node insertions in the second stage may cause the total number of nodes in the I-Tree to grow up to a value which is $\Omega(n^3)$ in the worst-case. This is illustrated in the following example–

EXAMPLE 4.2 Consider the following database (Table 4.2) containing $2m$ intervals, each of length $2m$. It is required to find the set of maximal $m$-
frequent intervals using Lin's method [Lin03]. The I-Tree constructed in the first stage of Lin's method is shown in Figure 4.2 and it has $O(m)$ nodes. Figure 4.3 shows the I-Tree after it is processed by the PT algorithm in the second stage and the number of nodes is now greater than $m^2$.

**Table 4.2 A database of intervals**

<table>
<thead>
<tr>
<th>No.</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>2m-1</th>
<th>2m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>$[1, 2m+1]$</td>
<td>$[2, 2m+2]$</td>
<td>...</td>
<td>$[2m-1, 4m-1]$</td>
<td>$[2m, 4m]$</td>
</tr>
</tbody>
</table>

![Diagram of I-Tree](image)

**Figure 4.2 The I-Tree constructed from Table 4.2 in the first stage of Lin's method**
**Time-complexity of Lin’s method**: The worst-case of Lin’s two-stage method for determining maximal $k$-frequent intervals has time-complexity $O(\max\{n \times (|LL| + |RR|), |LL| \times |RR|\})$. Here, $O(n \times (|LL| + |RR|))$ is for constructing the I-Tree in the first stage and $O(|LL| \times |RR|)$ is the time-complexity of the PT algorithm. Since $|LL|$ and $|RR|$ are less than or equal to $n$, both the stages of
Lin's two-stage method are $O(n^2)$. Also, since a sequential search is required to locate the point of insertion of every interval into the I-Tree, the worst-case time-complexity of the first stage is $\Omega(n^2)$. Example 4.2 shows that the number of nodes processed by the PT algorithm is $\Omega(n^2)$. Hence the worst-case time-complexity of the second stage is also $\Omega(n^2)$. Thus, in the worst-case, both the stages in Lin's method are $\theta(n^2)$.

4.4.2 Scope for improvement in Lin's method

In an experimental implementation of Lin's two-stage method, it was observed that a large fraction of the total computation time was consumed in the first stage itself—i.e. in the construction of the I-Tree from the database of intervals. A more time-efficient construction of the I-Tree may therefore reduce the computational effort of Lin's method. This possibility is explored in Section 4.5.

According to Lin, all intervals of the form $[l_i, r]$ with $r$ in $L_j$ for $j \leq i$ are potential maximal $k$-frequent intervals. To be able to compute the support of all such intervals, new nodes are inserted into the I-Tree in the second stage. The construction of the I-Tree in the first stage and then the insertion of new nodes into the I-Tree in the second stage, both lead to a $\Omega(n^2)$ worst-case time-complexity of Lin's method. An entirely different approach that abandons the use of the I-Tree structure may be able to improve the $\theta(n^2)$ worst-case time-complexity of Lin's method. This possibility is explored in Section 4.6.
4.5 THE PROPOSED IMPROVEMENT IN LIN’S METHOD BY INCREASING THE EFFICIENCY OF THE I-TREE CONSTRUCTION

The $\Theta(n^2)$ technique that is used to construct an I-Tree in the first stage of Lin’s method [Lin03] can be improved by defining the following total order $\leq$ among the intervals in the input database:

$$[a, b] \leq [c, d] \text{ if (i) } a < c \text{ or (ii) } a = c \text{ and } b \leq d$$

In this section, an algorithm called Build_ITree is proposed. The Build_ITree algorithm uses the order defined above and constructs an I-Tree from a database of $n$ intervals $[a_i, b_i]_{i=1}^n$ in $O(n)$ time after an initial $O(n \log n)$ sorting step.

Algorithm Build_ITree

Input:

$[a_i, b_i]_{i=1, 2, ..., n}$: A given database of $n$ intervals

Local variables:

$p$: header node containing the fields $p_{\text{left}}, p_{\text{right}}, p_{\text{count}}, p_{\text{next-head}}, p_{\text{next-node}}$ (these fields are described in Section 4.4.1)

$q$: non-header node containing the fields $q_{\text{left}}, q_{\text{right}}, q_{\text{count}}, q_{\text{next-node}}$ (these fields are described in Section 4.4.1)

$l_{\text{min}}$: the smallest left endpoint among the input intervals

$r_{\text{max}}$: the largest right endpoint among the input intervals

$j$: integer

$[l, r]$ : the last interval handled
Outputs

**H**: header list representing the I-Tree

**Step 1.** Sort \([a_i, b_i] \mid i = 1, 2, \ldots, n\) in non-decreasing order of \(\leq\), determine \(r_{max}\) and \(l_{min}\)

**Step 2.** \(l = l_{min} - 1\), \(r = r_{max} + 1\), \(H = \text{null} /*\ Initialization step */

**Step 3.** for \(j = 1\) to \(n\)

**Step 4.** if \((a_j > l)\) then /* A new header node is required */

**Step 4(a)** \(l = a_j\), \(r = b_j\)

**Step 4(b)** \(p = \text{new header node, } p_{\text{left}} = l, p_{\text{right}} = r_{max}, p_{\text{count}} = 0, p_{\text{next-node}} = \text{NIL}, p_{\text{next-head}} = \text{NIL},\) insert \(p\) at the end of \(H\)

**Step 4(c)** if \((b_j\) is equal to \(r_{max}\)) then

\[p_{\text{count}} = p_{\text{count}} + 1\]

else

\(q = \text{new header node, } q_{\text{left}} = a_j, q_{\text{right}} = b_j, q_{\text{count}} = 1,\) insert \(q\) after \(p\) in the non-header list that has \(p\) as the head

endif

**Step 4(d)** continue

endif

**Step 5.** if \((b_j > r)\) then /* A new header node is not required */

**Step 5(a)** \(r = b_j\)

**Step 5(b)** if \((b_j\) is equal to \(r_{max}\)) then
p_{count} = p_{count} + 1

else

q = new non-header node, q_{left} = a_j,
q_{right} = b_j, q_{count} = 1, insert q after p in
the non-header list that has p as the head

endif

Step 5(c) continue

endif

Step 6. if (b_j is equal to r_{max}) then

p_{count} = p_{count} + 1

else

q_{count} = q_{count} + 1

endif

end for

Step 7. return H

In the algorithm, a header node p, a non-header node q and the last interval
handled [l, r] are maintained. First, the intervals in the database are sorted in non-
decreasing order. The I-Tree is initially empty and is built up by inserting the
sorted intervals [a_j, b_j]_{j=1}^{n} one by one into the I-Tree. During the insertion of an
interval [a_j, b_j] into the I-Tree, if a_j > l, then at first, a new header node p
representing the interval [l, r_{max}] with p_{count} = 0 is inserted at the end of H. The
non-header list with p as the head is now appropriately updated. If on the other
hand, \( b_j > r \), then a new header node need not be created. Instead, the non-header list with the current header node \( p \) as the head is appropriately updated. If neither \( a_j > 1 \) nor \( b_j > r \), then we must have \( a_j = 1 \) and \( b_j = r \) (because the intervals are sorted in non-decreasing order) and so it suffices to increase either \( p_{\text{count}} \) or \( q_{\text{count}} \).

**Time-Complexity of Build_ITree**: Step 1, in which the \( n \) input intervals are sorted is \( O(n \log n) \). Step 2 is \( O(1) \). Since the input intervals are initially sorted, a sequential search is not required to locate the point of insertion of an interval into the I-Tree. As such, each iteration of the for loop (Step 3 to Step 6) is \( O(1) \). The insertion of the \( n \) input intervals into the I-Tree is hence \( O(n) \). Step 7 is \( O(1) \). Therefore, the overall worst-case time-complexity of the Build_ITree algorithm is \( O(n \log n) \).

**The contribution to Lin’s method**: The \( O(n \log n) \) Build_ITree algorithm that constructs an I-Tree from a database of \( n \) intervals is an improvement over the \( \Theta(n^2) \) I-Tree construction technique proposed by Lin [Lin03]. The experimental results given in Section 4.8.1 clearly show that the overall computational time taken by Lin’s method is considerably reduced if it uses the Build_ITree algorithm to construct the I-Tree from the input intervals.

**Please note**: In Lin’s method, the set of maximal k-frequent intervals is determined by the \( \Theta(n^2) \) PT algorithm after scanning the I-Tree constructed from the input intervals. As mentioned in Section 4.4.2, it was experimentally observed that a large fraction of the overall computational time of Lin’s method is taken up in the construction of the I-Tree from the input intervals. By using the proposed \( O(n \log n) \) Build_ITree algorithm to construct the I-Tree from the input intervals, the overall computational time of Lin’s method can be considerably reduced. The worst-
case time-complexity of Lin's method however still remains $O(n^3)$. An asymptotically faster method for determining maximal k-frequent intervals will be presented in the next section.

4.6 MIntMiner — A NEW ALGORITHM FOR MINING MAXIMAL K-FREQUENT INTERVALS

First, in Section 4.6.1, a framework of results related to k-frequent intervals and maximal k-frequent intervals is introduced. Then on the basis of this framework, a new algorithm MIntMiner is presented in Section 4.6.2 for determining the set of maximal k-frequent intervals with respect to an interval transaction database.

4.6.1 Theoretical framework

Please note: Theorem 4.1 actually follows from the corresponding result on maximal frequent sets, the proof of which is already known. However, for the sake of completeness, the proof of this result for maximal k-frequent intervals has been included.

In the following theorems and corollaries, S and S' are interval transaction databases.

Theorem 4.1 If $I \in F(k, S)$, then there is some $J \in M(k, S)$ such that $I \subseteq J$

Proof: The non-empty intersection of a finite number of intervals is an interval (result 3., Chapter III). Let $K$ be the set of all non-empty intervals of the form $\cap_{i=1}^{k} \text{intv}(T_i)$, where $T_1, T_2, \ldots, T_k$ are k interval transactions of S. Clearly $K$ is finite and every element of $K$ is k-frequent with respect to S. Now, let $I$ be a k-frequent interval with respect to S. Obviously $I \subseteq J_1$ for some $J_1 \in K$. If $J_1$ is not maximal k-frequent with respect to S, then by Definition (4.5), there is an
interval $I_1$ which is k-frequent with respect to $S$ and $I_1 \subseteq J_1$. Since $I_1$ is k-frequent with respect to $S$, $I_1 \subseteq J_2$ for some $J_2 \in K$. Continuing in this way, the following sequence of intervals is obtained,

$$I \subseteq J_1 \subseteq I_1 \subseteq J_2 \subseteq I_2 \subseteq J_3, \ldots \ldots \cdots \cdots$$ \hspace{1cm} (4.1)

Since $J_1 \in K$ and $K$ is finite, the sequence (4.1) cannot continue indefinitely and we shall obtain $J_m$ which is a maximal k-frequent interval with respect to $S$. Since $I \subseteq J_m$, this proves the theorem.

**Theorem 4.2** A maximal k-frequent interval with respect to $S$ is an interval and each of it's endpoints is an endpoint in $S$.

**Proof:** Let $I$ be a maximal k-frequent interval with respect to $S$. Then, there are k interval transactions $T_1, T_2, \ldots, T_k$ in $S$ that support $I$ i.e. each of $I_1 = \text{intv}(T_1)$, $I_2 = \text{intv}(T_2), \ldots, I_k = \text{intv}(T_k)$ contain $I$. Let

$$J = I_1 \cap I_2 \cap \ldots \cap I_k$$ \hspace{1cm} (4.2)

Now,

$$I \subseteq J$$ \hspace{1cm} (4.3)

Clearly,

$$\sigma(J, S) \geq k$$

and so $J$ is also k-frequent with respect to $S$. Now, it follows from Definition (4.5) that $I \not\subseteq J$. Hence (4.3) yields

$$I = J$$

33
Therefore, by (4.2), every maximal k-frequent interval is an intersection of k intervals \( \text{intv}(T_1), \text{intv}(T_2) \ldots \text{intv}(T_k) \) in \( S \). Result(3.) (Chapter III) now yields the desired result.

**Theorem 4.3** If \([L, R]\) and \([L', R']\) are two distinct elements of \( M(k, S) \), then either \( L < L' \) and \( R < R' \) or \( L' < L \) and \( R' < R \)

**Proof:** Since \([L, R]\) and \([L', R']\) are two distinct maximal k-frequent intervals with respect to \( S \), \( L \) and \( L' \) are distinct and \( R \) and \( R' \) are distinct. Otherwise, one of the intervals will be properly contained in the other, which contradicts the hypothesis that each interval is maximal. If \( L < L' \) and \( R' < R \), then \([L', R'] \subset [L, R]\)

which again contradicts the hypothesis that \([L, R]\) and \([L', R']\) are two distinct maximal k-frequent intervals with respect to \( S \). Hence, if \( L < L' \), then \( R < R' \).

Similarly, if \( L' < L \), then \( R' < R \). This proves the result.

**Corr. 4.3.1** If \( M(k, S) \) is non-empty, the members of \( M(k, S) \) can be arranged as \([L_1, R_1], [L_2, R_2], \ldots, [L_m, R_m]\) with \( L_1 < L_2 < \ldots < L_m \) and \( R_1 < R_2 < \ldots < R_m \)

**Proof:** Follows from Theorem 4.3.

**Please note:** Throughout this chapter, we shall refer to \([L_4, R_4]\) as first(\( M(k, S) \)).

**Theorem 4.4** If \( S' \subseteq S \), then \( F(k, S') \subseteq F(k, S) \)

**Proof:** Let \( I \in F(k, S') \). Then, there are at least \( k \) interval transactions in \( S' \) that support \( I \). By the hypothesis \( S' \subseteq S \), it is clear that these \( k \) interval transactions are in \( S \) also. Hence, \( I \in F(k, S) \). This proves the theorem.

**Theorem 4.5** If \( S' \subseteq S \), \( J \in M(k, S) \) and \( J \in F(k, S') \), then \( J \in M(k, S') \)
Proof. Suppose \( J \in M(k, S') \). Because of the hypothesis \( J \in F(k, S') \), by Theorem 4.1, it can be concluded that there is some \( K \) in \( M(k, S') \) such that

\[
J \subseteq K \tag{4.4}
\]

By Definition (4.5),

\[
M(k, S') \subseteq F(k, S')
\]

and this implies that

\[
K \in F(k, S')
\]

By Theorem 4.4, it can be concluded that

\[
K \in F(k, S) \tag{4.5}
\]

However, now the hypothesis \( J \in M(k, S) \) is contradicted by (4.4).

Hence, it can be concluded that \( J \in M(k, S') \) and this proves the theorem.

Corr. 4.5.1 If \( S' \subseteq S \), \( J \in M(k, S) \) and no interval transaction of \( S - S' \) supports \( J \), then \( J \in M(k, S') \)

Proof: Since no interval transaction of \( S - S' \) supports \( J \),

\[
\sigma(J, S) = \sigma(J, S') \tag{4.6}
\]

By Definition (4.5), the hypothesis \( J \in M(k, S) \) implies that

\[
J \in F(k, S)
\]

and so by (4.6), it can be concluded that

\[
J \in F(k, S')
\]

Now by Theorem 4.5, \( J \in M(k, S') \), which is the desired result.

Corr. 4.5.2 If \( S' \subseteq S \) and \( M(k, S) \subseteq F(k, S') \), then \( M(k, S) = M(k, S') \)

Proof: Let \( J \in M(k, S) \). The hypothesis \( M(k, S) \subseteq F(k, S') \) implies that

\[
J \in F(k, S')
\]
Hence, by Theorem 4.5, it can be concluded that

\[ J \in M(k, S') \]

So,

\[ M(k, S) \subseteq M(k, S') \quad (4.7) \]

Now, let

\[ J \in M(k, S') \]

From Definition (4.5), it is clear that

\[ J \in F(k, S') \]

By Theorem 4.4,

\[ J \in F(k, S) \quad (4.8) \]

Now, suppose

\[ J \notin M(k, S) \]

Since \( J \in F(k, S) \), by Theorem 4.1, it is clear that there is some \( K \) in \( M(k, S) \) such that

\[ J \subset K \quad (4.9) \]

The hypothesis \( M(k, S) \subseteq F(k, S') \) implies that

\[ K \in F(k, S') \quad (4.10) \]

However, now \( J \in M(k, S') \) is contradicted by (4.9) and (4.10)

Hence,

\[ J \in M(k, S) \]

and thus

\[ M(k, S') \subseteq M(k, S) \quad (4.11) \]

By (4.7) and (4.11),
\[ M(k, S) = M(k, S') \]

This proves the result.

**Corr. 4.5.3** If \( U \) is a set of interval transactions such that no interval transaction of \( U \) supports any member of \( M(k, S) \), then \( M(k, S) = M(k, S - U) \)

*Proof*: Let \( J \in M(k, S) \). So,

\[ \sigma(J, S) \geq k \]

From the hypothesis, it is clear that no interval transaction of \( U \) supports \( J \). Because of this,

\[ \sigma(J, S) = \sigma(J, S - U) \]

So,

\[ \sigma(J, S - U) \geq k \]

which means that

\[ J \in F(k, S - U) \]

Thus,

\[ M(k, S) \subseteq F(k, S - U) \]

Now, by Corr. 4.5.2,

\[ M(k, S) = M(k, S - U) \]

This proves the result.

**Theorem 4.6** Let an interval transaction \( T \in S \). If \( \text{intv}(T) \subseteq J \) for some \( J \in M(k, S) \) and \( S' = S - \{T\} \), then \( M(k, S) = M(k, S') \)

*Proof*: Let

\[ K \in M(k, S) \quad (4.12) \]

and suppose
K ⊆ intv(T)

The hypothesis intv(T) ⊆ J implies that

\[ K ⊆ J \]  \hspace{1cm} (4.13)

But by the hypothesis \( J \in M(k, S) \) and (4.12), it is clear that (4.13) is a contradiction. Therefore intv(T) does not contain any member of \( M(k, S) \) and so applying Corr. 4.5.3 for \( U = \{T\} \), it can be concluded that

\[ M(k, S) = M(k, S') \]

This proves the theorem.

**Corr. 4.6.1** Let \( U \) be a set of interval transactions such that for every interval transaction \( T \) of \( U \), intv(T) = I for a fixed interval I and I ⊆ J for some J in \( M(k, S) \). Then \( M(k, S) = M(k, S - U) \).

*Proof:* Follows by applying Theorem 4.6 to each interval transaction \( T \) of \( U \).

**Theorem 4.7** If \( S' \subseteq S \), \( J \in M(k, S') \) and no interval transaction of \( S - S' \) supports \( J \), then \( J \in M(k, S) \).

*Proof:* Since \( J \in M(k, S') \), it is clear from Definition (4.5) that,

\[ J \in F(k, S') \]

By Theorem 4.4,

\[ J \in F(k, S) \]  \hspace{1cm} (4.14)

Let \( U \) be an interval such \( J \subseteq U \). Then by Definition (4.5),

\[ \sigma(U, S') < k \]  \hspace{1cm} (4.15)

By the hypothesis, no interval transaction of \( S - S' \) supports \( J \) and so no interval transaction of \( S - S' \) supports \( U \). Because of this,

\[ \sigma(U, S) = \sigma(U, S') \]
and so by (4.15), it follows that,

\[ \sigma(U, S) < k \]  \hspace{1cm} (4.16)

Now, by (4.14) and (4.16),

\[ J \in M(k, S) \]

which proves the theorem.

### 4.6.2 Formalizing MIntMiner

Using the framework established in Section 4.6.1, the MIntMiner algorithm is now formalized. Section 4.6.2.1 explains the underlying principle of the algorithm. Section 4.6.2.2 describes the data-structures that are used. The pseudo-code description is provided in Section 4.6.2.3. This is followed by an example in Section 4.6.2.4, that illustrates the working of the MIntMiner algorithm. Section 4.6.2.5 and Section 4.6.2.6 provide the correctness and efficiency arguments respectively.

#### 4.6.2.1 Working Principle

Given an interval transaction database \( S_0 \) and a minimum absolute support \( k \), the proposed algorithm MIntMiner determines the set of maximal \( k \)-frequent intervals with respect to \( S_0 \). Throughout the algorithm, an interval transaction database \( S \) and a set \( MF \) of intervals are maintained. In the beginning, \( S \) is set to \( S_0 \) and \( MF \) is initialized to NIL. Thus in the beginning, \( M(k, S_0) = M(k, S) \cup MF \) holds. Henceforth, whenever any maximal \( k \)-frequent interval with respect to \( S \) is discovered, it is added to \( MF \). Also, interval transactions in \( S \) are progressively removed from \( S \). The MIntMiner algorithm
however ensures that after every change of \( S \) and / or \( MF \), \( M(k, S) \cup MF \) does not change. Therefore, the relation \( M(k, S_0) = M(k, S) \cup MF \) is an invariant of the algorithm. At termination, \( M(k, S) \) becomes empty and hence \( M(k, S_0) = MF \) at termination.

### 4.6.2.2 Data-structures Used

Two data-structures are used by the MIntMiner algorithm:

(i) \[
\text{endpt} \{\text{domain_element } x; \\
\text{int } b;\}
\]

An \textit{endpt} node stores information about an endpoint; it has two fields: \( x \) and \( b \); \( x \) is a domain element (a user-specified data-type whose underlying domain has a total order \( \leq \)); \( b \) is an integer used to represent a bracket type viz. ‘[’ or ‘]’. The endpt nodes can be ordered using the total order \( \leq \) defined in the set of endpoints (see Definition 3, Chapter III).

(ii) \[
\text{epp} \{\text{int } freq, \text{isdeleted}; \\
\text{endpt } e; \\
\text{epp* } pe;\}
\]

An \textit{epp} node captures information about a set of interval transactions having the same interval \( I \). An epp node has four fields: \( freq, isdeleted, e \) and \( pe \); \( freq \) gives the number of interval transactions in the set; \( isdeleted \) is a flag that indicates whether this set of interval transactions has been deleted from some given interval transaction database; \( e \) is a node of type
endpt and it represents one of the endpoints of I; pe gives the address of the epp node whose e field represents the other endpoint of I.

A pair of epp nodes can therefore be used to represent any set of interval transactions having the same interval I. e.g. Suppose 8 interval transactions have the interval [4, 8]. These 8 interval transactions can be represented by a pair of epp nodes, say $e_1$ and $e_2$, in the following manner:-

$$e_1.freq = 8;\ e_1.e = (4,');\ e_1.pe = \text{address of } e_2;$$
$$e_2.freq = 8;\ e_2.e = (8,']');\ e_2.pe = \text{address of } e_1.$$  

**Please note:** The value of the `isdeleted` field of the epp nodes $e_1$ and $e_2$ will depend on whether this set of 8 interval transactions has been deleted from some given interval transaction database.

A total order $\leq$ is defined among epp nodes as follows: Let ep1 and ep2 be two epp nodes. Now,

$$ep1 \leq ep2 \text{ if } ep1.e < ep2.e$$

or

$$\text{if } ep1.e = ep2.e \text{ and } ep1->pe->e \leq ep2->pe->e$$

The relation $\leq$ is clearly reflexive and transitive. Also clearly, if $ep1.e = ep2.e$ and $ep1->pe->e = ep2->pe->e$, then ep1 and ep2 will refer to the same interval and so $ep1->freq = ep2->freq$. Hence, $ep1 = ep2$. Because of this, the relation is anti-symmetric. Hence $\leq$ is a partial order defined among the epp nodes. The fact that the order is total is clear from the definition.
4.6.23 Pseudo-code

Global variables defined and set by the Prepare_eppnodes algorithm:

n: integer /* number of distinct intervals in S₀ */
ep[]: array of pointers to epp nodes

Algorithm Prepare_eppnodes(S₀, N)

Input:

S₀: interval transaction database
N: integer /* number of interval transactions in S₀ */

Output:

None

Global variables updated:

n and ep[]

/* Given below are the steps of the algorithm Prepare_eppnodes */

Step 1. create_eppnodes(S₀)

/* This routine uses a pair of epp nodes to represent each of the interval transactions in S₀; except for the field isdeleted, all the other fields of the epp nodes are set; pointers to these epp nodes are inserted in the array ep[]. The pseudo-code of the routine is given in Appendix II. */

Step 2. sort(N)
/* This routine sorts the array ep[1, 2N] created in Step 1 in non-decreasing order of *(ep[i]) */

Step 3. n = compress(N)

/* This routine uses a single pair of *(ep[i]) to represent all the interval transactions in S₀ that have the same interval. The routine is thus essentially an elimination of duplicates in the sorted array ep[1...2N]. However the freq and pe fields of *(ep[i]) need to be adjusted appropriately. The routine returns the number of distinct intervals in S₀. The pseudo-code of the routine is given in Appendix II */

/* After executing the Prepare_eppnodes algorithm, ep[1...2n] – a sorted array of pointers to 2n epp nodes representing S₀ is obtained. This will now be used by the MIntMiner algorithm to determine the set of maximal k-frequent intervals with respect to S₀ */

Algorithm MIntMiner(k)

Input:

k: integer /* minimum absolute support */

Output:

M(k, S₀): The set of maximal k-frequent intervals with respect to the interval transaction database S₀ that was input to the Prepare_eppnodes algorithm

Global variables accessed:
Global variables updated:

ep[] /* only the isdeleted field of the epp nodes is updated */

/* An interval transaction database $S$ is maintained by the MIntMiner algorithm. $S$
 is represented collectively by *(ep[i]) (i = 1 to 2n)* having ep[i]->isdeleted = 0.
 It may be noted that ep[i]->isdeleted = 0 if and only if ep[i]->e is an endpoint
 in $S$. Otherwise, ep[i]->isdeleted = 1.
 */

Variables in the scope of the MIntMiner algorithm:

lc: integer /* array index */

rc: integer /* array index */

p: integer /* used to keep track of the total number of left endpoints in $S$ that
 are less than or equal to ep[lc]->e */

MF: set of intervals

j: integer /* loop counter */

/* delintv(i): Whenever this routine is invoked, ep[i]->e is guaranteed to be a right
 endpoint of $S$ (i.e. ep[i]->e.b = ']' and ep[i]->isdeleted = 0). This routine is used
 to remove from $S$, the set of interval transactions that are represented by the
 following pair of epp nodes – *(ep[i]) and *(ep[i]->pe)
 */

delintv(i): void

Parameter.
i: integer
{
    if (ep[i]->pe->e ≤ ep[lc]->e) then
        ρ = ρ - ep[i]->freq
    endif
    ep[i]->isdeleted = 1
    ep[i]->pe->isdeleted = 1
}

/* slide_to_right(): This routine is invoked by locate_left() if ep[lc+1]->e is a left
endpoint in S. This routine is used by locate_left() to increase the value of lc
and adjust the value of ρ accordingly..*/
slide_to_right(): void
{
    pc, lcc: integer
    pc = ρ
    lcc = lc
    lcc = lcc + 1
    pc = pc + ep[lcc]->freq
    while (ep[lcc+1]->e is equal to ep[lcc]->e) /* Note that ep[lcc]->e is now a
        left endpoint */
        lcc = lcc + 1
    if (ep[lcc]->isdeleted is equal to 0) then
\[ pc = pc + ep[lcc]->freq \]

endif

end while

\[ p = pc \]

\[ lc = lcc \]

} 

*/ locate_left(): This routine is invoked by find_first(). Whenever this routine invoked, \( p < k \). If \( M(k, S) \) is empty, locate_left() returns 0. Otherwise it returns 1 and sets \( lc \) to the largest value satisfying \( ep[lc]->e = (a,'\)\), where \( (a,'\)\) is the left endpoint of first(\( M(k, S) \)) (see comments following Corr. 4.3.1) */

locate_left(): integer

{

repeat

if (lc is equal to 2n) then

return 0

endif

if (ep[lc + 1]->isdeleted is equal to 1) then

lc = lc + 1

continue

endif

if (ep[lc + 1]->e is a left endpoint) then

slide_to_right()

}
if \((p \geq k)\) then
    return 1
endif
else /* ep[lc+1] is a right endpoint */
    increment lc and call delintv(lc)
endif
forever

locate_right(): void
{
    if (rcdc) then
        rc = lc
    endif
    repeat
        rc = rc + 1
        if (ep[rc]->e is a left endpoint) then
            continue
        endif
        if \((\rho < k + ep[rc]->freq\) and \(ep[rc]->pe->e < ep[lc]->e\)) \) then
            \(47\)
return
endif
delintv(rc)
forever
}
/* find_first(): If M(k, S) is empty, this routine returns 0. Otherwise, it returns 1 and
if [a, b] is first(M(k, S)), the routine sets lc to the largest value satisfying
ep[lc]->e = (a, '[') and rc to a value satisfying ep[rc]->e = (b, ']') */
find_first(): integer
{
if (locate_left() is equal to 0) then
    return 0
endif
locate_right()
return 1
}
/* extract_first(): This routine is invoked when M(k, S) is non-empty. If
first(M(k, S)) = [a, b], then when this routine is invoked, lc is at the largest value
satisfying ep[lc]->e = (a, '[') and rc is at a value satisfying ep[rc]->e = (b, ']').
The routine inserts first(M(k, S)) = [a, b] into the set MF and removes from S,
al\text{all interval transactions whose right endpoint e satisfies e.x = b}. It also sets
rc to the largest value satisfying ep[rc]->e = (b, ']')

extract_first(): void
{
    a, b: domain element /* [a, b] is first(M(k, S)) */
    a = ep[lc]->e.x
    b = ep[rc]->e.x
    MF = MF \cup [a, b]
    delintv(rc)
    while (rc < 2n and ep[rc+1]->e is equal to (b, ')))
        rc = rc + 1
        delintv(rc)
    end while
}

/* Given below are the steps of the algorithm MIntMiner */

Step 1. lc = rc = p= 0, MF = NIL /* initialization step */
Step 2. for j = 1 to 2n /* initialization step */
    ep[j]->isdeleted = 0
end for
Step 3. while (find_first() is equal to 1)
    extract_first()
end while
Step 4. return MF

At first, the Prepare_eppnodes algorithm is used to obtain a sorted array ep[1...2n] of pointers to 2n epp nodes that collectively represent all the interval transactions in S₀. The MIntMiner algorithm next uses this ep[1...2n] array (which is sorted in non-decreasing order of *(ep[i])) to determine the set of maximal k-frequent intervals with respect to S₀. Throughout the MIntMiner algorithm, a set MF of intervals and an interval transaction database S are maintained. S is collectively represented by *(ep[i]) (i= 1 to 2n) having ep[i]->isdeleted = 0. It is to be noted that ep[i]->isdeleted = 0 if and only if ep[i]->e is an endpoint in S. Otherwise, ep[i]->isdeleted = 1. At the start of the MIntMiner algorithm, MF is initialized to NIL and S is set to S₀. The find_first() routine looks for first(M(k, S)) via two other routines – viz. locate_left() and locate_right(). The locate_left() routine uses the array index variable lc to search for the left endpoint of first(M(k, S)). If the left endpoint of first(M(k, S)) is detected i.e. if M(k, S) is non-empty, then locate_right() uses the array index variable rc to locate the right endpoint of first(M(k, S)). It may be noted that under different conditions, locate_left() and locate_right() may also reduce S by invoking the delintv(i) routine. If first(M(k, S)) is successfully located by find_first(), find_first() returns 1 and the extract_first() routine is then invoked. The extract_first() routine inserts first(M(k, S)) into the set MF and then reduces S. When the find_first() routine fails to locate first(M(k, S)) i.e. when S (and hence M(k, S)) is empty, find_first() returns 0. In this case, the MIntMiner algorithm terminates and
returns MF. At termination, MF gives the required set of maximal k-frequent intervals with respect to $S_0$.

It may be noted that to determine $M(k, S_0)$ for different values of $k$, it suffices to execute the Prepare_eppnodes algorithm just once. After that, only the MIntMiner algorithm needs to be invoked, once for each value of $k$.

4.6.2.4 An Example

EXAMPLE 4.3 Determine the set of maximal 3-frequent intervals with respect to the interval transaction database (Table 4.3) given below:

Table 4.3 An interval transaction database

<table>
<thead>
<tr>
<th>tid</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>[1, 2]</td>
<td>[2, 6]</td>
<td>[3, 6]</td>
<td>[3, 5]</td>
<td>[3, 9]</td>
<td>[4, 5]</td>
<td>[8, 10]</td>
<td>[8, 9]</td>
<td>[8, 9]</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>

Solution: At first, the Prepare_eppnodes algorithm is executed to obtain the sorted array ep[1...18] of pointers to 18 epp nodes that represent the input interval transaction database. It may be recalled here that the ep[1...18] array is sorted in non-decreasing order of *(ep[i]) and that the ep[i]->isdeleted field of *(ep[i]) is yet to be set. The MIntMiner algorithm now starts. Table 4.4 illustrates *(ep[i]) (i = 1 to 18) after Step 2 (i.e. after the initialization steps) of the MIntMiner algorithm. At this point, it is seen that S is the input interval transaction database. It may be recalled here that the MIntMiner algorithm maintains an interval transaction database S and S is represented
by *(ep[i]) (i = 1 to 2n) having ep[i]->isdeleted = 0. Also at this point, lc = rc = p = 0 and MF = NIL.

Table 4.4 Status of some variables after Step 2 of the MIntMiner algorithm is executed on the data in Table 4.3

|   | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ep[i]->isdeleted | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| ep[i]->pe         | 3  | 10 | 1  | 8  | 11 | 15 | 9  | 4  | 7  | 2  | 5  | 16 | 17 | 18 | 6  | 12 | 13 | 14 |
| ep[i]->freq       | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 2  | 1  | 1  | 1  | 2  | 1  | 1  |

Please note: ep[i]->pe gives the array index j such that ep[j]->e is the partner endpoint of ep[i]->e.

Table 4.5 traces all the iterations of the while loop in Step 3 of the MIntMiner algorithm. In each row of Table 4.5, it is shown how the values of lc, rc, p and MF change after the action(s) (as listed in that row) have been executed. Table 4.6 illustrates how S gets progressively reduced after each call to delintv(j) in the MIntMiner algorithm. It may be recalled here that set of interval transactions represented by *(ep[i]) (i = 1 to 2n) having ep[i]->isdeleted = 1 are no longer in S.
Table 4.5 Trace of the while loop iterations in Step 3 of the MIntMiner algorithm for the data in Table 4.3

<table>
<thead>
<tr>
<th>Action(s)</th>
<th>lc</th>
<th>rc</th>
<th>p</th>
<th>MF</th>
</tr>
</thead>
<tbody>
<tr>
<td>find_first() starts</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>NIL</td>
</tr>
<tr>
<td>locate_left() starts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() starts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc, p increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() starts</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>NIL</td>
</tr>
<tr>
<td>lc, p increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>NIL</td>
</tr>
<tr>
<td>delintv(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p reduced by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() starts</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>NIL</td>
</tr>
<tr>
<td>lc, p increased by 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locate_left() ends and</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>returns 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locate_right() starts</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>NIL</td>
</tr>
<tr>
<td>rc set to lc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delintv(8)</td>
<td>p reduced by 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc increased by 1</td>
<td>6</td>
<td>9</td>
<td>3</td>
<td>NIL</td>
</tr>
<tr>
<td>delintv(9)</td>
<td>rc increased by 1</td>
<td>6</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>locate_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find_first() ends and returns 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>extract_first() starts</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>[3, 6]</td>
</tr>
<tr>
<td>a = 3, b = 6</td>
<td>[3, 6] added to MF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delintv(10)</td>
<td>p reduced by 1</td>
<td>6</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>rc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delintv(11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p reduced by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>extract_first() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find_first() starts</td>
<td>11</td>
<td>11</td>
<td>1</td>
<td>[3, 6]</td>
</tr>
<tr>
<td>locate_left() starts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

54
<table>
<thead>
<tr>
<th>lc increased by 1</th>
<th>13</th>
<th>11</th>
<th>4</th>
<th>[3, 6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>slide_to_right() starts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p increased by 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locate_left() ends and returns 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locate_right() starts</td>
<td>13</td>
<td>15</td>
<td>3</td>
<td>[3, 6]</td>
</tr>
<tr>
<td>rc set to lc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delintv(15)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p reduced by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rc increased by 1</td>
<td>13</td>
<td>16</td>
<td>3</td>
<td>[3, 6]</td>
</tr>
<tr>
<td>locate_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find_first() ends and returns 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>extract_first() starts</td>
<td>13</td>
<td>16</td>
<td>1</td>
<td>[3, 6], [8, 9]</td>
</tr>
<tr>
<td>a = 8, b = 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[8, 9] added to MF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delintv(16)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p reduced by 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>extract_first() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Value1</td>
<td>Value2</td>
<td>Value3</td>
<td>Value4</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>find_first() starts</td>
<td>14</td>
<td>16</td>
<td>2</td>
<td>[3, 6], [8, 9]</td>
</tr>
<tr>
<td>locate_left() starts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() starts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc, p increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide_to_right() ends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td>16</td>
<td>16</td>
<td>2</td>
<td>[3, 6], [8, 9]</td>
</tr>
<tr>
<td>lc increased by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td>17</td>
<td>16</td>
<td>1</td>
<td>[3, 6], [8, 9]</td>
</tr>
<tr>
<td>delintv(17)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p reduced by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc increased by 1</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>[3, 6], [8, 9]</td>
</tr>
<tr>
<td>delintv(18)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p reduced by 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locate_left() ends and returns 0</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>[3, 6], [8, 9]</td>
</tr>
<tr>
<td>find_first() ends and returns 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The algorithm MIntMiner terminates.
Table 4.6 Reduction of S after each call to delintv(j) for the data in Table 4.3

<table>
<thead>
<tr>
<th>After delintv(3)</th>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After delintv(8)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After delintv(9)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After delintv(10)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After delintv(11)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After delintv(15)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After delintv(16)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>After delintv(17)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>After delintv(18)</td>
<td>i</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>ep[i]-&gt;isdeleted</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

As shown in Table 4.5, when the MIntMiner algorithm terminates, MF = \{[3, 6], [8, 9]\} and this is the required set of maximal 3-frequent intervals with respect to the given interval transaction database (Table 4.3).

4.6.2.5 Correctness Claims

For proving the correctness of the MIntMiner algorithm, at first, definitions
4.6 and 4.7 and lemmas 4.1 to 4.14 (Please refer to Appendix III for the detailed proofs) are introduced.

Please note: $S_0, S, n, N, k, ep[i], lc, rc, MF$ and $p$ have been defined earlier in Section 4.6.2.3. The same definitions are associated with them throughout this section also.

**DEFINITION 4.6** $U(i)$. $U(i)$ is defined to be the set of interval transactions represented by *(ep[i]) and *(ep[i]->pe). If ep[i]->isdeleted = 1 and ep[i]->pe->isdeleted = 1 i.e. if the set of interval transactions represented collectively by *(ep[i]) and *(ep[i]->pe) is not in $S$, then $U(i)$ is considered to be empty. In this case, $|U(i)| = 0$. If ep[i]->isdeleted = 0 and ep[i]->pe->isdeleted = 0 i.e. if the set of interval transactions represented collectively by *(ep[i]) and *(ep[i]->pe) is in $S$, then $U(i)$ is non-empty and $|U(i)| = ep[i]->freq$.

**DEFINITION 4.7** $I(i)$. $I(i)$ is the interval associated with $U(i)$.

**Lemma 4.1** If $i \neq j$, $U(i) = U(j)$ if and only if ep[i]->pe = ep[j] (in which case ep[j]->pe = ep[i]). Otherwise, $U(i)$ and $U(j)$ are disjoint.

**Lemma 4.2** $lc \leq 2n$ and for every $i < lc$, if $U(i)$ is non-empty, then ep[i]->e is a left endpoint i.e. if $(b, 'l')$ is a right endpoint in $S$, then $b \geq ep[le]->e.x$.

**Lemma 4.3** $p = \sum_{i=1}^{lc} |U(i)|$

**Lemma 4.4** The value of $p$ is less than $k$ whenever locate_left() is invoked.

**Lemma 4.5** $M(k, S)$ remains unchanged by locate_left().
Lemma 4.6 If \( ep[lc] \rightarrow e \cdot x = a \) when slide_to_right() exits, then \( p = \sigma([a, a], S) \).

Lemma 4.7 If locate_left() returns 0, then \( M(k, S) \) is empty. If locate_left() returns 1, then \( M(k, S) \) is non-empty and locate_left() sets \( lc \) to the largest value satisfying \( ep[lc] \rightarrow e = (a, \['') \), where \( (a, \['') \) is the left endpoint of \( \text{first}(M(k, S)) \).

Lemma 4.8 At the beginning of an iteration of the repeat-forever loop in locate_right(), if the following conditions are satisfied:

\begin{itemize}
    \item[a)] \( M(k, S) \) is not empty and \( ep[lc] \rightarrow e \) is the left endpoint of \( \text{first}(M(k, S)) \)
    \item[b)] \( rc \leq 2n \)
    \item[c)] \( ep[i] \rightarrow e \) is a left endpoint for every non-empty \( U(i) \) with \( i \leq rc \) then
        \begin{itemize}
            \item[(i)] condition b) holds good throughout the iteration (and hence at the start of the next iteration, if any)
            \item[(ii)] condition c) holds good at the beginning of the next iteration, if any
            \item[(iii)] \( M(k, S) \) is not changed by the iteration
            \item[(iv)] condition a) holds good throughout the iteration (and hence at the start of the next iteration, if any)
        \end{itemize}
\end{itemize}

Lemma 4.9 At the beginning of the first iteration of the repeat-forever loop in locate_right(), all the conditions of Lemma 4.8 - viz. conditions a), b) and c) hold
Lemma 4.10 M(k, S) is not changed by locate_right()

Lemma 4.11 Whenever ep[m] is accessed, m is within bounds i.e. 1 ≤ m ≤ 2n

Lemma 4.12 When locate_right() exits, ep[rc]->e is the right endpoint of first(M(k, S))

Lemma 4.13 If find_first() returns 0, then M(k, S) is empty. If find_first() returns 1, then M(k, S) is non-empty and find_first() sets lc and rc such that lc is the largest value satisfying ep[lc]->e = (a, '[') and rc is a value satisfying ep[rc]->e = (b, ']'), where [a, b] = first(M(k, S))

Lemma 4.14 The routine extract_first() inserts first(M(k, S)) into MF and deletes some interval transactions from S to get S' such that M(k, S') = M(k, S) - first(M(k, S))

Theorem 4.8 The MIntMiner algorithm runs without any abnormal termination and determines all and only the maximal k-frequent intervals with respect to the given interval transaction database $S_0$

Proof: An abnormal termination of the MIntMiner algorithm can occur only when an access of the ep[] array goes out of bounds. Hence, by Lemma 4.11, it is established that the MIntMiner algorithm does not terminate abnormally. When the MIntMiner algorithm starts, S is set to $S_0$ and MF is initialized to NIL. Thus in the beginning, $M(k, S_0) = M(k, S) \cup MF$ holds. The find_first() routine does not update MF. Also, by Lemma 4.5 and Lemma 4.10, it follows that $M(k, S)$ remains unchanged by the find_first() routine. The find_first()
routine therefore does not change \( M(k, S) \cup MF \). By Lemma 4.14, it follows that the \texttt{extract\_first()} routine does not change \( M(k, S) \cup MF \). The relation \( M(k, S_0) = M(k, S) \cup MF \) is therefore an invariant of the MIntMiner algorithm. The MIntMiner algorithm terminates when \texttt{find\_first()} returns 0 and at this point, \( M(k, S) \) is empty (Lemma 4.13). Hence, when the MIntMiner algorithm terminates, \( M(k, S_0) = MF \) and this establishes the correctness of the MIntMiner algorithm.

4.6.2.6 Efficiency Arguments

\textit{Worst-case time-complexity of the Prepare\_eppnodes algorithm}: If \( n \) is the total number of interval transactions in the input interval transaction database, then in the worst-case, Step 1 and Step 3 are \( O(n) \) and Step 2 is \( O(n \log n) \). Hence, the overall worst-case time-complexity is \( O(n \log n) \).

\textit{Worst-case time-complexity of the MIntMiner algorithm}: For an algorithm expressed in a structured language, the worst-case time-complexity is given by \( O(n_f + n_w) \), where \( n_f \) is the total number of function calls in the algorithm and \( n_w \) is the total number of iterations of the various loops in the algorithm. Let \( n \) be the total number of interval transactions in the input interval transaction database. Since the values of \( lc \) and \( rc \) are always within bounds (Lemma 4.11), there are \( O(n) \) different values of \( lc + rc \). In the MIntMiner algorithm, the value of \( lc + rc \) is always non-decreasing and at the beginning of two successive calls of the same function, the value of \( lc + rc \) is always different. At the beginning of a call to any two different functions also, the value of \( lc + rc \) is always different. Thus \( n_f \) is \( O(n) \). The
for loop in Step 2 is obviously $O(n)$. In a particular call to `slide_to_right()`, the value of $lc$ is increased by at least the number of iterations of the while loop and hence the number of iterations of this loop is $O(n)$. At the beginning of two successive iterations of any other loop in the algorithm, the value of $lc + rc$ is always different and so the total number of iterations of these loops is also $O(n)$. Thus $n_w$ is $O(n)$. The overall worst-case time-complexity of the MIntMiner algorithm is hence $O(n)$.

To determine $M(k, S_0)$ for a different values of $k$, the `Prepare_eppnodes` algorithm has to be executed just once. After this, the MIntMiner algorithm needs to be executed once for each value of $k$. Thus, the overall worst-case time-complexity of this task is $O(n(a + \log n))$.

4.7 DETERMINING WHETHER A GIVEN NON-EMPTY INTERVAL IS K-FREQUENT

**Theorem 4.9** Suppose $[L_i, R_i]$, $i = 1, 2, \ldots, m$ is the set of maximal k-frequent intervals with respect to an interval transaction database such that $L_1 < L_2 < \ldots < L_m$ and $R_1 < R_2 < \ldots < R_m$. The necessary and sufficient conditions for a non-empty interval $[a, b]$ to be k-frequent with respect to the given interval transaction database are

(i) $L_1 \leq a$ and

(ii) If $(L_i, \ l^r)$ is the largest left endpoint less than or equal to $(a, \ l^r)$, then $b \leq R_i$

**Proof:** Suppose $[a, b]$ is k-frequent with respect to a given interval transaction database. By Theorem 4.1, for some $j$, $1 \leq j \leq m$,
So,
\[ L_j \leq a \]  \hspace{1cm} (4.18)

By Corr. 4.3.1, (4.18) implies
\[ L_1 \leq a \]

Hence (i) is true. Further, if \((L_i, \lceil \cdot \rceil)\) is the largest left endpoint less than or equal to \((a, \lceil \cdot \rceil)\), then by (4.18),
\[ L_j \leq L_i \]

Because of the hypothesis, this implies
\[ R_j \leq R_i \]  \hspace{1cm} (4.19)

By (4.17) and (4.19),
\[ b \leq R_j \leq R_i \]

Hence (ii) is true. Conversely, if (i) is true, then \(L_1 \leq a\). Let \((L_i, \lceil \cdot \rceil)\) be the largest left endpoint less than or equal to \((a, \lceil \cdot \rceil)\). Since (ii) is true, \(b \leq R_i\).

Hence,
\[ [a, b] \subseteq [L_i, R_i] \]

and therefore \([a, b]\) is k-frequent with respect to the given interval transaction database. This proves the theorem.

**Algorithm Chk_Freq**

**Input:**

\([a, b]\): interval

\([L_i, R_i]_{i=1,\ldots,m}\) with \(L_1 < L_2 < \ldots < L_m\) and \(R_1 < R_2 < \ldots < R_m\): the set of maximal k-frequent intervals with
Output:

isfrq: integer

/* The algorithm returns 1 if \([a, b]\) is \(k\)-frequent with respect to the given interval transaction database. Otherwise, it returns 0. */

Step 1. if \((a < L_i)\) then
    return 0
endif

Step 2. Using binary search, find the largest \(L_i \leq a\)

Step 3. if \((b \leq R_i)\) then
    return 1
else
    return 0
endif

Correctness claims: The correctness of the Chk_Freq algorithm follows from Theorem 4.9.

Worst-case time-complexity of the Chk_Freq algorithm: Let \(m\) be the total number of maximal \(k\)-frequent intervals with respect to the given interval
transaction database. Step 1 and Step 3 are $O(1)$ and Step 2 is $O(\log m)$. Thus, the overall worst-case time-complexity of the Chk_Freq algorithm is $O(\log m)$.

4.8 EXPERIMENTAL RESULTS

In Section 4.8.1, the performances of Lin’s method [Lin03] for mining maximal k-frequent intervals, the modified Lin’s method (proposed in Section 4.5) and the MIntMiner algorithm (proposed in Section 4.6) are compared. In Section 4.8.2, the performance of the Chk_Freq algorithm (proposed in Section 4.7) is evaluated. All the algorithms have been implemented in C++ on a Intel Core2Duo 3.06 Ghz processor PC with 1 GB of memory running Debian Linux 6.03. For all the algorithms, the computational time (in seconds) is used as the performance metric. The computational timings reported reflect the algorithmic performance only and do not include the input and the output time.

For the purpose of the experiments, a data-generator has been developed to generate test interval transaction databases. The data-generator program uses the following user-specified parameters –

- $n$: the total number of interval transactions
- $m$: mean of interval length
- $l_{\text{max}}$: the largest left endpoint

To generate an interval transaction database containing $n$ interval transactions having left endpoints distributed uniformly between 1 and $l_{\text{max}}$. The lengths of the intervals in the interval transaction database are in Poisson distribution with mean $m$. 

65
4.8.1 Lin's method Vs. Modified Lin's method Vs. MIntMiner

Using different parameter settings, the data generator first creates several test interval transaction databases—all exhibiting different characteristics. These test interval transaction databases are now used to study the comparative performances of the three algorithms/methods as also to observe the impact of minimum absolute support and different data characteristics on the performances of the algorithms/methods.

Four types of experiments are conducted. For each type, the value of one of the following four inputs viz. \(n\), \(m\), \(l_{\text{max}}\) and \(\text{minsup} \) (minimum absolute support \(\div n\)) is varied, while keeping the other three constant. For each type of experiment that is carried out, the algorithms/methods are tested on three sets of values of the inputs that are held constant.

*Type I Experiments (Figures 4.4 through 4.6):* Here; the effect of \(n\) – viz. the size of the input interval transaction database – on the performances of Lin’s method, the modified Lin’s method and MIntMiner is studied.
Figure 4.4 \( \ell_{\text{max}} = 800, \ m = 500, \ \text{minsup} = 0.05 \)
Figure 4.5 $l_{max} = 850$, $m = 450$, $\minsup = 0.1$
Figure 4.6  \( I_{\text{max}} = 900, m = 400, \text{minsup} = 0.05 \)
Empirical observations: The performance trends of the three algorithms/methods under study are similar for the range of the values of \( n \) that has been used. Apart from some minor fluctuations, the computational time increases linearly with \( n \).

Type II Experiments (Figure 4.7 through 4.9): Here, we study the effect of the mean of interval length \( m \) on the performances of Lin's method, the modified Lin's method and MIntMiner.
Figure 4.7  $l_{max} = 800$, $n = 200 K$, $minsup = 0.05$
Figure 4.8 \( l_{\text{max}} = 850, n = 300 \text{ K}, \text{ minsup} = 0.1 \)
Figure 4.9 \( l_{max} = 900, n = 400 \ K, \minsup = 0.05 \)
Empirical observations: The performance trends of the three algorithms/methods under study are similar for the range of the values of m that has been used. Apart from some minor fluctuations, the computational time remains constant with the increase of m.

Type III Experiments (Figures 4.10 through 4.12): Here, the effect of minimum absolute support on the performances of Lin’s method, the modified Lin’s method and MIntMiner is studied.
Figure 4.10  $l_{\text{max}} = 800$, $n = 200K$, $m = 400$
Figure 4.11 $l_{\text{max}} = 850$, $n = 300$ K, $m = 450$
Figure 4.12 $l_{\text{max}} = 900$, $n = 400,000$, $m = 500$
Empirical observations: The performance trends of the three algorithms/methods under study are similar for the range of values of minimum absolute support that has been used. Apart from some minor fluctuations, the computational time remains constant with the increase of minimum absolute support.

Type IV Experiments (Figures 4.13 through 4.15): Finally, the effect of $l_{\text{max}}$ – viz. the largest left endpoint in an interval transaction database – on the performances of Lin's method, the modified Lin's method and MIntMiner is studied.
Figure 4.13\ n = 200 K, m = 400, \text{mins}up = 0.05
Figure 4.14  n = 300 K, m = 450, minsup = 0.1
Figure 4.15  \( n = 400 \text{ K}, m = 500, \text{ minsup} = 0.05 \)
Empirical observations: For the range of values of lmax that has been used, apart from some minor fluctuations,

- the computational time of Lin's method increases linearly with the value of lmax
- the computational time of the modified Lin's method and the MIntMiner algorithm remains constant with the increase of lmax.

Overall observations: The modified Lin's method consistently outperforms Lin's method by a factor of at least 4. In most of the cases, the improvement is at least by a factor of 7. The MIntMiner algorithm consistently outperforms the modified Lin's method by a factor of at least 5.

4.8.2 Evaluation of the Chk_Freq algorithm

The Chk_Freq algorithm is tested on all the intervals in a test interval transaction database TKTEST that is generated by the data-generator with the following parameters: n = 1000K, lmax = 800 and m = 20. TK1 is another interval transaction database generated with parameters: n = 10K, lmax = 200 and m = 400. For values of minimum absolute support k ranging from 2050 down to 1920 in steps of 10, Chk_Freq uses M(k, TK1) to check if all the intervals in TKTEST are in F(k, TK1). In Figure 4.16, for each value of k, |M(k, TK1)| is plotted against the total computational time taken by Chk_Freq to check if all the intervals in TKTEST are in F(k, TK1). Note that for the different values of k that are considered here—viz. 2050 down to 1920 in steps of 10, the values of |M(k, TK1)| are different. The total computational time that is reported does not
include the time taken by the MLntMiner algorithm to obtain \( M(k, TK1) \). Also, as mentioned earlier in the beginning of Section 4.8, the computational time does not include the input and the output time.

![Graph showing growth of total computational time](image)

**Figure 4.16 Growth of total computational time of the Chk Freq algorithm with \( |M(k, TK1)| \), for k varying from 2050 down to 1920 in steps of 10**

**Empirical observations:** As expected (from the time-complexity discussion presented in Section 4.7), the nature of the growth of the total computational time of the Chk Freq algorithm with \( |M(k, TK1)| \), for k varying from 2050 down to 1920 in steps of 10, is logarithmic.

### 4.9 SUMMARY OF CONTRIBUTIONS

At first, in this chapter, an earlier method proposed by Lin [Lin03] for mining maximal k-frequent intervals was modified. Experimental results showed that the
modified Lin's method consistently outperforms the original Lin's method [Lin03] by a factor of at least 4. In most of the cases, the improvement is at least by a factor of 7. Next, a framework of results related to maximal k-frequent intervals was introduced and based on this framework, an entirely new and more time-efficient algorithm MIntMiner was proposed for mining the set of maximal k-frequent intervals. A detailed proof of correctness of the MIntMiner algorithm was provided and its worst-case time-complexity was worked out. Experimental results showed that MIntMiner consistently outperforms the modified Lin’s method by a factor of at least 5. Finally, an algorithm Chk_Freq was proposed that efficiently uses the set of maximal k-frequent intervals to determine whether any given non-empty interval is k-frequent. The correctness of the Chk_Freq algorithm was established and its worst-case time-complexity was worked out. Experiments were performed to confirm the theoretical expectations of the algorithm’s time-efficiency.

Though the algorithm Chk_Freq determines whether any given non-empty interval is k-frequent, it cannot determine the value of the interval's absolute support. This problem will be addressed in the next chapter.