Object	timestamp	events
A	10	2, 3, 5
A	20	6, 1
A	23	1
B	11	4, 5, 6
B	17	2
B	21	7, 8, 1, 2
B	28	1, 6
D	14	1, 8, 7

Figure 1.18: Example Input Data

1.6 Bringing in the Sequential Relationships

The data collected from scientific experiments, or monitoring of physical systems such as telecommunications networks, or from transactions at a supermarket, have inherent sequential nature to them. Sequential nature means that the events occurring in such data are related to each other by relationships of the form *before* (or *after*) and *together*. The concept of item-sets and association rules discussed so far takes into account only the *together* part of the relationship, the information provided by the *before/after* relationships is ignored. This information could be very valuable in finding more interesting patterns hidden in the data, which could be useful for many purposes such as prediction of events or identification of better sequential rules that characterize different parts of the data.

In this section, we discuss the concept of *sequential associations*, more commonly known as *sequential patterns*, and algorithms to discover them. Before doing so, let us see some examples of the kind of information sequential patterns provide.

- In a computer bookstore, let one of the patterns discovered be (Introduction_to_perl HTML_1.0) (Web Development_Primer) (Python_Made_Easy). Here, assume that the books within a pair of brackets are bought together, whereas the items in consecutive brackets are bought within 3 days time. This pattern tells a lot about a customer's behavior, that the non-sequential association rule or frequent item-set cannot. Specifically, the pattern can be used to trigger the inventory check procedure after the books Introduction_to_Perl and HTML_1.0 have been sold to make sure that the next book in sequence, Web_Development_Primer, would be available within 3 days time.

- Gene Expression Example.

1.6.1 Nature of Input Dataset

The input to the problem of discovering sequential associations is a sequence data characterized by three columns: *object*, *timestamp*, and *events*. Each row records occurrences of events on an object at a particular time. An example is shown in Figure 1.18. Alternative way to look at the input data is in terms of the timeline representations of all objects (illustrated in Figure 1.22). It should be noted that the approach taken in [45] uses the same format of input data, but approach taken in [46] allows specifying only one object.

Various definitions of *object* and *events* can be used, depending on what kind of sequences one is looking for. For example, in one formulation, object can be a telecommunication switch, and event can be an alarm type occurring on the switch. With this, the sequences discovered will indicate interesting patterns of occurrences of alarm types occurring at a switch. In another formulation, object can be a *day* and event can be a switch or a switch-alarm type pair. This will give interesting sequential relations between different switches or switch-alarm type pairs over a day.

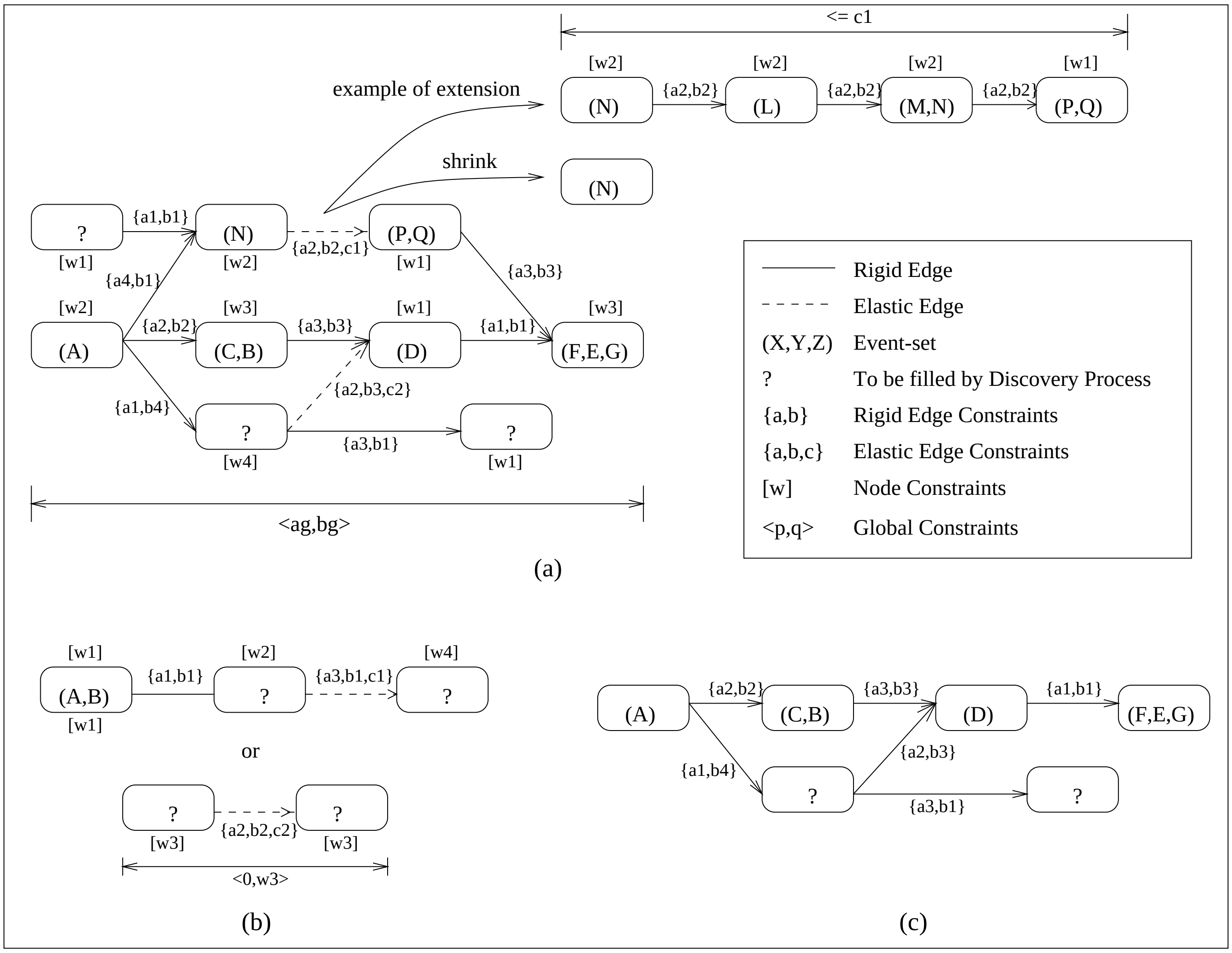


Figure 1.19: (a) The most universal formulation of a sequential relationship, (b) Single path formulations, (c) Formulation due to [BWJ96].

1.6.2 Formulation: Representation and Constraints

The sequential relationships among events can be expressed in various forms. The most general form of a valid sequential relationship can be represented by a directed acyclic graph (Figure 1.19(a)). A node would represent an event or a set of events. Some nodes will be associated with an event-set before the discovery process, which we call *event constraints*. Other nodes would be associated with different possible event-sets during the discovery process. A directed edge from node A to node B would indicate that events of A occur before events of B. A set of numbers is associated with each edge, which we call *edge constraints*. These constraints indicate the allowed separation between the occurrences of events of its incident nodes. For example, each edge can have a 2-element set $\{a,b\}$ associated with it, such that for an edge (A,B) –> (C,D), a represents the minimum required separation between latest event of the (A,B) set and the earliest event of the (C,D) set. On the other hand, b represents the maximum allowed difference in the earliest occurring event among (A,B) and the latest occurring event among (C,D). Each node may also be associated with a set of numbers, called *node constraints*, which governs the definition of when a set of events can be associated with a node; for example, if there is one number $[w]$ associated with a node, then the set of events (C,D) can be associated with that node only if C and D occur within w units of each other. Other than the edge and node constraints, there is one more set of constraints, called *global constraints*, which is imposed on the entire pattern. For example a constraint of the form $< ag,bg >$ imposed on the entire pattern may mean that the total duration of the pattern has lower limit of ag and upper limit of bg. Final characteristic of this dag-based sequential relationship is that each edge can belong to one of the two types: elastic or rigid. An elastic edge can be extended into multiple edges by adding nodes dynamically in succession during the discovery process, as shown in Figure 1.19(a). Each of the newly added edges would have same edge constraints as the original elastic edge, and each of the newly added nodes would have same node constraints as those of the starting node (A in case of an edge from A to B). The extension capability of an elastic edge is limited by a third constraint on the elastic edge, which limits the entire duration of the extended edge. This is denoted by c in the triplet $\{a,b,c\}$ associated with each elastic edge. An elastic edge can also be shrunk by collapsing one of its incidence nodes onto the other. A rigid edge cannot be changed in this manner during the discovery process.

No work has been done so far on the most general form of the sequential relationship described above. However, simplified versions of this dag have been used in the literature. Two prominent approaches have been taken so far for identifying sequential relationships. One approach, shown in Figure 1.19(b), restricts the nature of the relationship to a single path of the dag as presented in [45] and [46]. In both these works, there is only one elastic edge in the dag, and there are no event constraints. The second approach taken in [47] assumes the dag to have single root node (no incoming edges) and the structure of the dag is assumed to be fixed during the discovery process; i.e. all the edges are rigid. Figure 1.19(c) represents this approach. This approach allows event constraints. It also allows different edge constraints to be specified in different time units (or granularities).

Although the dag-based approach described in Figure 1.19(a) is the most generic in nature, it can be looked at as a compact way of representing multiple single-path sequential relationships. More precisely, a dag can be broken into all its constituent single path relationships by enumerating all the paths between roots and leaves. Root here is a node with no incoming edges, and leaf is a node with no outgoing edges. With this viewpoint, the algorithm which discovers the relationship represented by a dag can be visualized as multiple passes of an algorithm which discovers single path relationships. With this argument, there is no loss of generality if only the relationship represented by a single path of the dag is considered as the universal formulation. However, it should be pointed out that discovering relationships directly in the form of a dag can be more efficient than discovering relationships along all such individual paths, because it has a potential to avoid the repitition of work that would be incurred if individual paths are discovered independently. For the purposes of illustrating the issues of this paper, we assume that the dag is broken into all its constituent paths (or chains). *We refer to such single path relationships with all its event, node, edge, and global constraints, and two types of edges (rigid and elastic) as the univeral sequential pattern.* The node, edge, and global constraints are together refered to as *timing constraints*. It should be noted that all edge constraints are specified in same units (or granularity). Multiple granularities as used in [47] could be converted to single (finest) granularity before the discovery algorithm.

Before describing the details of universal sequential patterns, it might be worthwhile to note the differences and similarities between the approaches taken in [45, 46] in the framework of the general form representation that was presented above. As was noted earlier, the approaches are similar in the sense that they both discover relationships along a single path, and they both do not have any event constraints. The formulations, however, differ in two aspects. One is their ability to specify timing constraints, and the other is how they count the occurrences of a candidate pattern in a given dataset. The difference in the counting part will be elucidated when we present our unifying formulation in following sections, but the difference in the timing constraints can be made clear by Figure 1.20. The approach taken in [45] has no global constraints, whereas the approach taken in [46] has no edge or node constraints.

Universal Formulation in Detail

The universal formulation and an example of discovered pattern are given in Figure 1.21.

A universal sequential pattern is represented as a sequence of the form (A) (C,B) (D) (F,E,G). Here A,

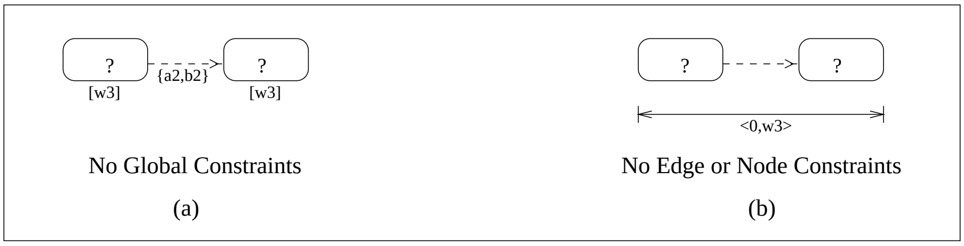


Figure 1.20: Comparing Timing Constraints in sequential pattern formulations of (a) Generalized Sequential Patterns due to [45], and (b) Episodes due to [46].

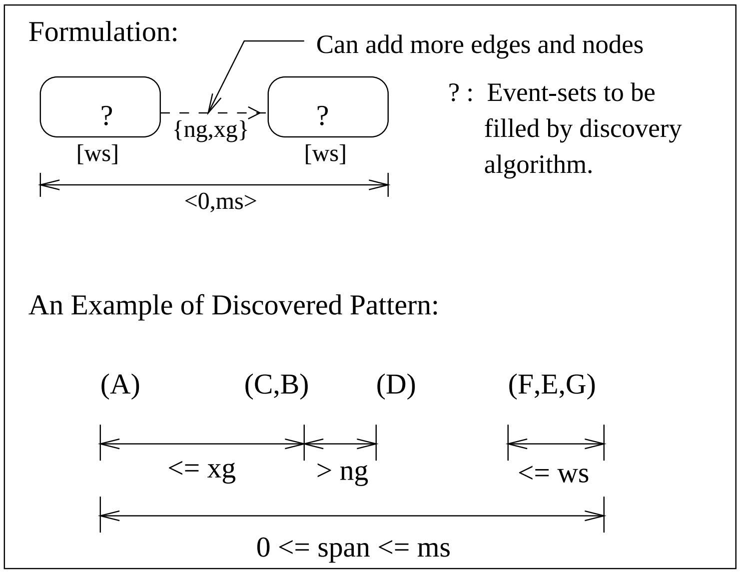


Figure 1.21: Universal Formulation of Sequential Patterns

B, etc. are the *events* that are *timestamped*. They can be telecommunication alarm types, switch-alarm type pairs, or items bought by a customer in a transaction, or anything that can have a timestamp. Here timestamp is used as a generic term to denote a measure of temporal nature. The example sequence above says that occurrence of event A is followed by occurrence of *event-set* (C,B) which is followed by occurrence of event D, and so on.

The occurrence of particular events can be fixed beforehand (event constraints). For the discovery process to be meaningful, at least one node should have its event-set unspecified. With event constraints present, the discovery algorithm can be optimized to reduce the search space, but in the remainder of discussion, we assume that there are no event constraints.

The node, edge, and global constraints translate into various input parameters which govern the times at which the events in a sequence can occur. Figure 1.21 illustrates these parameters, called timing constraints.

Here is the description of the parameters:

- ms **: Maximum Span** : The maximum allowed time difference between the latest and earliest occurrences of events in the *entire* sequence.
- ws **: Event-set Window Size** : The maximum allowed time difference between the latest and earliest occurrences of events in any *event-set*.
- xg **: Maximum Gap** : The maximum allowed time difference between the latest occurrence of an event in an event-set and the earliest occurrence of an event in its immediately preceding event-set.

- ng : **Minimum Gap** : The minimum required time difference between the earliest occurrence of an event in an event-set and the latest occurrence of an event in its immediately preceding event-set.

It can be seen that as far as timing constraints are concerned, above formulation is similar to the formulation in [45] if $ms \to \infty$; and it is similar to the episode formulations in [46]: serial episodes if $xg \geq ms$ and $ng = 0$ are used with ws such that each event-set has only one event, and parallel episodes if $ws = ms$, $xg \geq ms$, $ng \geq ms$. Actually, for the formulation to be exactly equivalent to those in [45] or [46], the choice of counting method also matters, which is discussed in the next section.

1.6.3 What is an interesting sequential pattern?

A sequence is said to be *interesting* if it occurs *enough* number of times satisfying the given timing constraints (ms, ws, xg, ng).

There are two issues here. First issue is, how the sequence occurrences are counted. This deals with different counting methods, and we will shortly elaborate on it in great detail. The second issue is, how many occurrences are *enough*? This is determined by the *support threshold*, which is an input parameter. After doing the counting, the sequences which do not occur *enough* number of times are filtered out. The support threshold can be specified in terms of absolute count, or percentage with respect to some basis. The support threshold is commonly used as a measure of interestingness of a pattern because of one of its very important properties. The property is, a subsequence of any sequence has at least as much support as the sequence, or put another way, the support for a sequence cannot be any greater than the support of any of its subsequences. This property helps to reduce the algorithmic complexity of discovering interesting patterns by allowing a systematic growth of patterns based on the apriori principle [45].

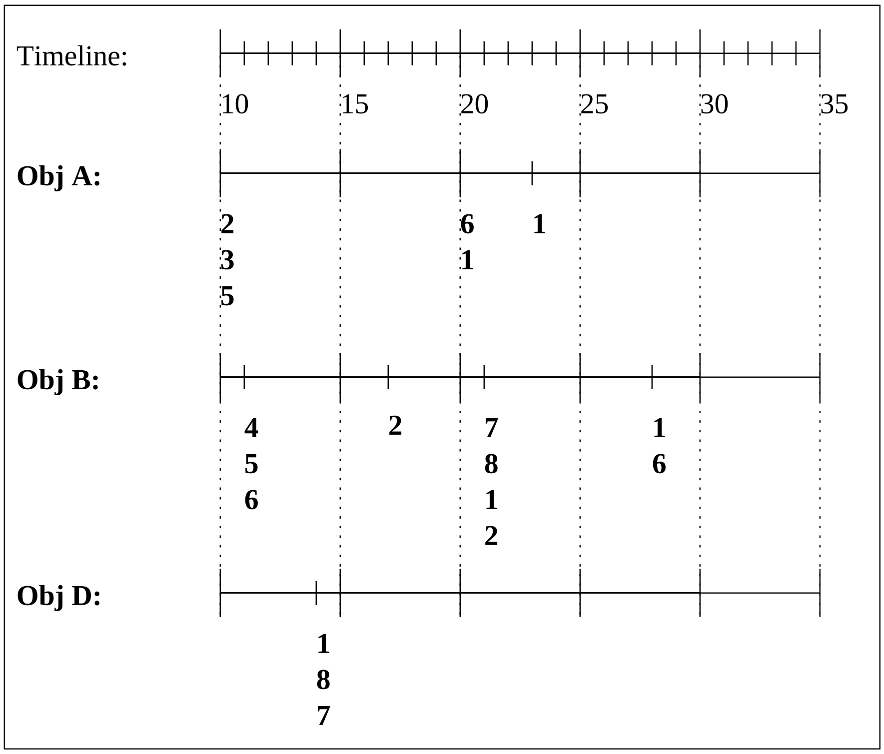
Coming back to the issue of *how-to-count*, there are five different ways defined for counting the number of occurrences. These can be divided into three conceptual groups.

First group, which consists of COBJ, just looks for an occurrence of a given sequence in an object's timeline. One occurrence is enough to ensure that the sequence occurs in that object. Second group is based on counting the windows in which the given sequence occurs. This consists of CWIN and CMINWIN. Third group is based on counting the distinct occurrences of a sequence. This consists of CDIST and CDIST_O. Although the occurrences are restricted to be contained in a window of size ms (maximum span), this group is different from the window-based counting group, because occurrences describe a cause-effect relationship directly based on the events themselves. The window-based group takes a different approach. It is based on the premise that the user is observing events occurring in a window, and is looking for the relationship to be exhibited in the window at least once.

This difference will become clearer as we describe the methods below. Figure 1.22 is used to illustrate their definition, whereas Figure 1.23 will elucidate the difference between different counting groups. Which approach is suitable will depend on the specific application that the user has in mind, and on user's domain expertise in the area. The applications in section 1.6.6 will throw more light on this aspect.

- (counting method = **COBJ**) One occurrence per object.
  The count here indicates the number of objects in which the sequence appears. For example, sequence (2) (1,6) with ms=20, ws=0 has two occurrences: for A, (2) at t=10 and (1,6) at t=20, and for B, (2) at t=17 and (1,6) at t=28. Note that, although (2) (1,6) appears in B one more time with (2) at t=21 and (1,6) at t=28, it is counted only once.

- (counting method = **CWIN**) One occurrence per span-window.
  Span-window is defined as a window of duration equal to span (ms). Consecutive span-windows have one time unit's difference in their respective start and end times. They move across the entire time duration of each object, but none of the span-windows spans across two different objects. The counts for all the objects are added up. In Fig. 3, with ms=20 and ws=5, sequence (2) (1,6) has 23 occurrences: it appears in 10 windows for A, with windows from t=1 to t=10, and in 13 windows for B, with windows from t=9 to t=21. Note that, the window of span 20 starting at t=9 is defined as the [9,29) interval.



(b)

Figure 1.22: Time-line representation of input data of Figure 1.18

- (counting method = **CMINWIN**) Number of Minimal Windows of Occurrence.
  A minimal window of occurrence is the smallest window in which the sequence occurs given the timing constraints. In other words, a minimal window is the time interval such that the sequence occurs in that time interval, but it does not occur in any of the proper subintervals of it. This definition can be considered as a restrictive version of CWIN, because its effect is to shrink and collapse some of the windows that are counted by CWIN.

  This method is similar in spirit to the concept of minimal occurrences in [46], but there is one difference. In [46], there is no duration limit on the minimal occurrence. Whereas, in our CMINWIN method, the size of the minimal window is limited by the maximum span (ms) constraint.

  All the minimal windows of occurrences are counted for each object, and then they are added up over all objects. For example, with ms=20, sequence (2) (1,6) has only two minimal window occurrences, one for A and one for B. The occurrence in B with (2) at t=16 and (1,6) at t=28 is not a minimal window occurrence because it contain another smaller window of occurrence with (2) at t=21 and (1,6) at t=28, which indeed is a minimal window of occurrence. It can be seen that many windows that were counted by CWIN method are collapsed into these two windows. For example, all those windows in object A that had the pattern same pattern (2)(1,6), with (2) at t=10 and (1,6) occurring in t=[20,23], are shrunk and collapsed into one minimal window [10,20].

- (counting method = **CDIST_O**) Distinct Occurrences with Possibility of Event-Timestamp Overlap.

  An distinct occurrence of a sequence is defined to be the set of event-timestamp pairs that satisfy the specified timing constraints, such that there has to be at least one new event-timestamp pair different from the previously counted occurrence. Counting all such distinct occurrences results in CDIST_O method.

  The number of occurrences counted using this method depends on the direction in which an object's timeline is scanned. We assumed that the timeline is scanned in the direction of increasing timestamps.

  As with all previous methods, the sequence occurrence must have all its events happening on the same object. All occurrences of the sequence are added over all objects.

  As an example, with ms=20, sequence (2) (1,6) has three distinct occurrences when overlap is allowed, one for A and two for B. The occurrence in B with (2) at t=21 and (1,6) at t=28 is a distinct occurrence because (2) at t=21 is the new event-timestamp pair from the previously counted (2) at t=17 and (1,6) at t=28 occurrence.

- (counting method = **CDIST**) Distinct Occurrences with No Event-Timestamp Overlap Allowed.

  In CDIST_O above, two occurrences of a sequence were allowed to have overlapping event-timestamp pairs. In this CDIST method, we don't allow any such overlap. So, effectively when an event-timestamp pair is considered for counting some occurrence of a sequence, it is flagged off and is never again considered for counting occurrences of that particular sequence for that particular object.

  As in CDIST_O, the sequence occurrence must have all its events happening on the same object, all occurrences of the sequence are added over all objects, and the timeline is scanned in forward direction.

  As an example, with ms=20, sequence (2) (1,6) has only two distinct occurrences, one for A and one for B. The occurrence in B with (2) at t=21 and (1,6) at t=28 is not a distinct one because events 1 and 6 at t=28 are already used in counting the first occurrence which has (2) at t=17.

The relationships and differences between methods CWIN, CDIST, CDIST_O, and CMINWIN are further clarified by Figure 1.23, which assumes presence of only one object. If n_O indicates the number of occurrences counted with method O, then it can be noted that $n_{CMINWIN} < n_{CWIN}$ and $n_{CDIST} < n_{CDIST\_O}$. Look at relationships across the columns of Figure 1.23(a). It can be noted that CWIN and CDIST_O are similar in spirit because they count *all* the windows and occurrences, respectively, and CMINWIN and CDIST are similar because they count the *minimal* windows or occurrences. The cause-effect philosophy used in the occurrence based approaches and the observation-window philosophy taken by window based approaches clearly yield different occurrence counts as illustrated in part (b) of Figure 1.23.

One final point that should be noted regarding the counting methods is as follows. If the percentage based support threshold is used, then we need to determine the basis for this support percentage. It depends on the counting method above is used. For the method COBJ, the basis is total number of objects in the input data. For methods CWIN and CMINWIN, the basis is the sum of the total number of span-windows possible in all objects. For methods CDIST and CDIST_O, the basis is maximum number of *all possible* distinct occurrences of a sequence over all objects, which is the number of distinct timestamps present in the input data of each object.

With this set of different counting methods, it is instructive to see how the existing formulations of sequential patterns map to the universal formulation. As stated in the introduction, the universal sequential patterns actually unify and generalize the notions of generalized sequential patterns (GSP) proposed in [45] and episodes proposed in [46], both of which can be shown to be the special cases of the universal formulation of Figure 1.21. If the maximum span constraint is considered ineffective ($ms \rightarrow \infty$) and COBJ method is used for counting, then the formulation is identical to GSP. If constraint $xg \geq ms$ and the CWIN counting method are used, then the formulation is equivalent to the *episodes* of [46]. In fact, for algorithmic convenience, the generic notion of episodes is broken down into two special kinds of episodes: serial and parallel. In addition to $xg \geq ms$ constraint and CWIN counting method, if we impose $ng = 0$ and set ws such that each event-set is restricted to have only one event, then the universal formulation becomes equivalent to serial episodes. On the other hand, if additional constraints are set to $ws = ms$ and $ng \geq ms$, then the formulation is equivalent to the parallel episodes.

There are a few other formulations of sequential patterns proposed in the literature [47, 48]. In terms of representation capability, they can be shown to be the special cases of the sophisticated version of universal sequential patterns given in Figure 1.19(a). The formulation of [47] is equivalent to a dag that has a rigid structure, only the ng and xg timing constraints, and stricter event constraints. This formulation was shown earlier in Figure 1.19(c). The formulation given in [48] is based on regular expressions (RE). The deterministic finite automaton representation of their formulation can be shown to be a special case of the dag-based universal sequential pattern. Also, each of the paths of this automaton can be represented by the universal sequential patterns of Figure 1.21.

1.6.4 What is an interesting sequential rule ?

Section 1.6.3 discussed interesting sequential patterns. For each interesting sequential pattern discovered, a sequential rule predicting the occurrence of last event-set in the sequence can be deduced. For example, for a pattern (A) (B C), the rule deduced is of the form, If A occurs, then event-set (B,C) occurs within the timing constraints specified. This rule is represented as (A) $\rightarrow$ (B,C). Similarly, for a pattern (A,B) (C) (D), the rule predicting occurrence of D after occurrence of (A,B) (C), is formed as (A,B) (C) $\rightarrow$ (D).

Stating it formally, rule S1 $\rightarrow$ IS, predicting IS, is formed using the sequential pattern, S : S1 (IS), where S1 is the subsequence formed by omitting the last event-set, IS. We say that this rule is *interesting* or *strong* if occurrence of S1 predicts occurrence of IS with a large *significance*. Significance is defined as Confidence / (co(IS) / Support basis), where Confidence is in turn defined as co(S) / co(S1). Here, co(S), co(S1) and co(IS) denote the number of occurrences of sequential patterns S, S1, and IS, respectively. Obviously, the same method must be used for counting occurrences of S, S1, and IS. Confidence is a number less than 1, and significance can be any number greater than 0.

Intuitively, confidence tells the conditional probability with which one can predict the occurrence of the consequent, IS, after seeing an occurrence of the antecedent sequence, S1. A rule which occurs with high confidence and relatively less occurrences of the consequent, together implying a high significance, is *strong* because the antecedent predicts *most* of the consequent's occurrences with high confidence. There is another characteristic of the rule called *coverage* that tells us about the fraction of times the entire sequential pattern occurs with respect to the number of times the consequent occurs. Coverage is defined as co(S) / co(IS). A rule with high significance and high coverage has a better predictive power.

(a)

	Count All	Count Minimal
Count Windows	CWIN	CMINWIN
Count Occurrences	CDIST_O	CDIST

(b)

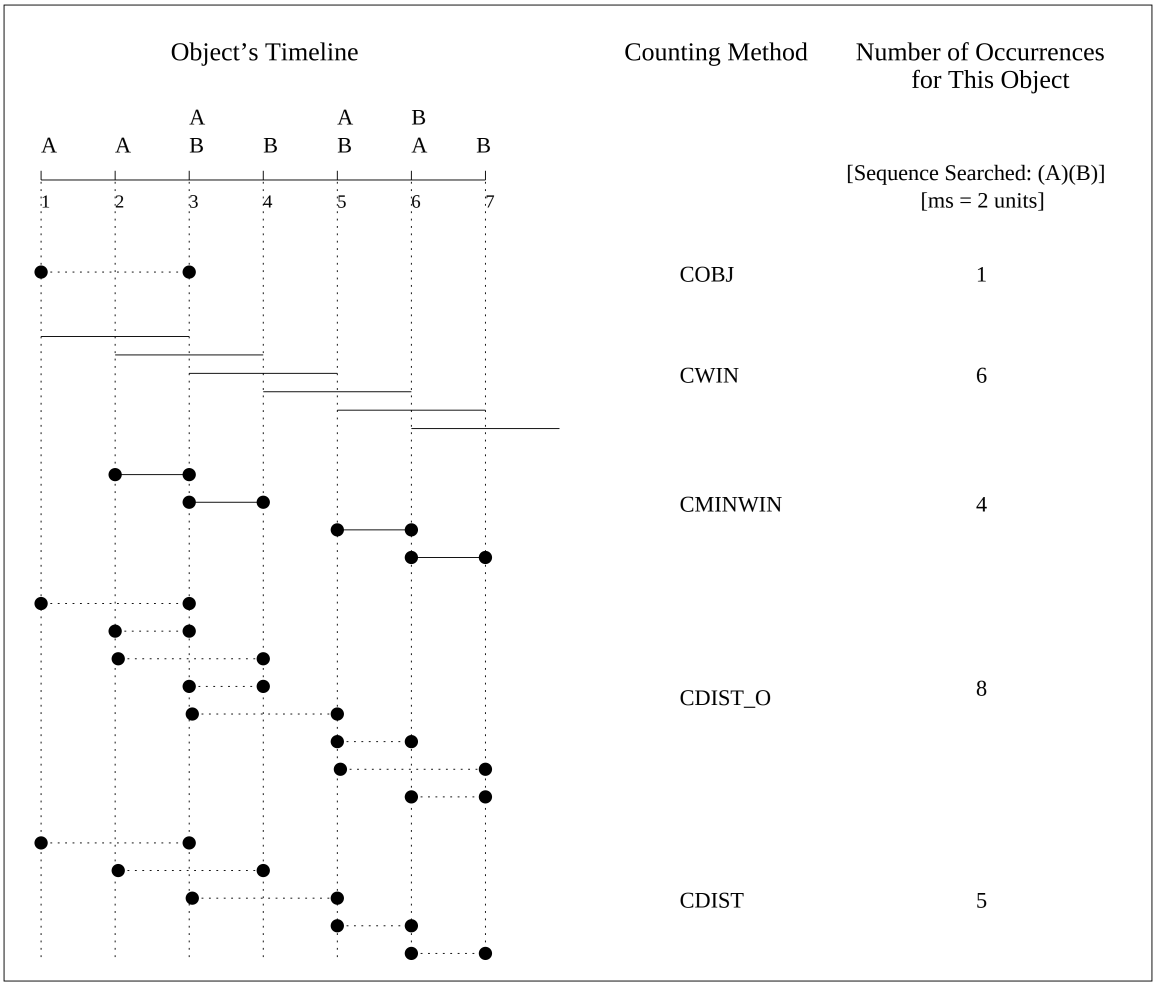


Figure 1.23: Comparing different counting methods. (a) Relationship among methods that count multiple occurrences per object (all but COBJ), (b) Differences among methods.

When the sequential pattern has only one event-set in it, the rule formed will predict the occurrence of these events happening within the ws duration. In this case, the confidence, significance, and coverage numbers associated with the rule are formed by averaging the corresponding numbers over all the rules that predict occurrence of each of the events in the event-set. As an example, if the pattern is (A,B,C), then three rules will be formed (A,B) $\rightarrow$ (C), (B,C) $\rightarrow$ (A), and (A,C) $\rightarrow$ B; the significance, confidence, and coverage are computed for each of these three rules and then they are averaged out.

Once the measures of interestingness are computed for all the discovered rules, they can be ordered in any way the user wishes. Usually, an ordering that lists the rules in decreasing order of significance will list interesting and strong rules at the top. Ties can be broken using confidence, coverage, and support in that order.

1.6.5 Discovery of Universal Sequential Patterns

The complexity of discovering frequent sequences is much more than the complexity of mining non-sequential associations. The reason is that, the maximum number of sequences having k events is $O(m^k 2^{k-1})$, where m is the total number of distinct events in the input data. In contrast, there are only $\binom{m}{k}$ possible item-sets of size k, when there are total of m distinct items. Using the definition of interestingness of a sequence, and the timing constraints imposed on the events occurring in a sequence, many of these sequences can be pruned. If a sequence does not occur sufficient number of times or if it does not satisfy the timing constraints, it will not be present in the final result. But in order to contain the computational complexity, the search space needs to be traversed in a systematic manner that searches only those sequences that would potentially satisfy both the support and timing constraints. As was done in the Apriori algorithm for non-sequential associations, the sequence discovery process can be guided by using the monotonicity property of support of a sequence. This property implies that as we add more events to the sequence, its support will decrease monotonously. So, if the frequent sequences having only one event are found first, then these can be used to build potential sequences having two events. All events occurring in such potential sequences are themselves frequent. Taking this notion further in an inductive manner, the frequent sequences having $k-1$ events can be used to build *potential* frequent sequences having k events. A potential k-sequence will be such that all its $(k-1)$-subsequences are themselves frequent.

The procedure outlined above is used in the GSP algorithm [45], but it discovers restrictive sequential patterns defined in [45]. We present below, a modified GSP algorithm that discovers patterns according to our universal formulation defined in previous subsection.

Similar to GSP, the algorithm works in iterations over the number of events in the sequence. In every iteration it has two phases. The first phase of join-and-prune generates the potentially frequent k-sequences, called candidates, from frequent $(k-1)$-sequences. The second phase counts the occurrences of candidates in object time-lines, and forms the set of frequent k-sequences, which seeds the next iteration. The structure of this algorithm is given in Figure 1.24.

As an illustration of the join-and-prune phase of the algorithm, consider the following example. Let the set of frequent 2-sequences be $F_2 = \{(1)(2), (1)(3), (1\ 3), (2\ 3), (3)(4)\}$. Now, in the join phase, (1)(2) will join with (2 3) to give a potential 3-sequence (1)(2 3). This sequence will not be pruned away because all of its 2-subsequences (1)(2), (1)(3), and (2 3) are in F_2. Now, consider the join between (1)(3) and (3)(4). This will generate a sequence (1)(3)(4), but it will be pruned away because (1)(4) is not present in F_2. This example assumes that the xg constraint is absent. If xg was specified, it would not be correct to prune (1)(3)(4) based on the absence of (1)(4), because it is not necessary for (1)(4) to satisfy the xg timing constraint in order for (1)(3)(4) to satisfy it.

The candidates that are generated after the join-and-prune phase need to searched in the input sequence data to count their occurrences. One simple way is to scan each object once for the occurrence of each candidate. This may become very expensive if the number of candidates is large, and if the number of of events occurring on input objects is large. Another possibility is to generate all the k-sequences present in all the span-size windows of an object's timeline and see if they are present in the set of candidates generated. This approach can also be very expensive especially because it ignores the information obtained from the previous pass that counted $(k-1)$-sequences in the timeline. Such information is implicitly stored in the

Structure of the Algorithm: (Based on GSP [45])

```
form Set of Frequent Sequences, F_1 each having 1 event;
k = 2;
while ( F_{k-1} is not empty )
    C_k = join-and-prune(F_{k-1}). C_k is the set of Candidates having k events
            [Store C_k in a hash tree access structure]

    for each object's timeline
        repeat
            [hash-tree-traversal]
                stream the timeline through hash tree to search presence
                of possible candidates, until leaf node is encountered; ⟵
            [leaf-count]
                count the occurrences of candidates at leaf nodes;        ⟵
        until (no more traversing is possible)
    end

    form F_k from C_k by retaining only the large candidates (having count above
    support threshold);
end

function join-and-prune(F_{k-1}): returning C_k
begin
    if (k = 2) then
        for every pair of events, e_1 and e_2 in F_1,
            form c = (e_1)(e_2); add c to C_k
        for every pair of events, e_1 and e_2 in F_1, such that e_1 < e_2
            form c = (e_1 e_2); add c to C_k
    else
        for every c_1 in F_{k-1}
            search c_2 in F_{k-1} such that
                deleting first event of c_1 and last event of c_2 yield same subsequences
            for every such c_2, form k-sequence c such that
                c = (c_1)(e) if (e) is the last event-set of c_2; otherwise c = (c_1 e)
                    [e is the last event of c_2]
                if every (k - 1) subsequence of c is present in F_{k-1}, then
                    add c to C_k
        end
    end
end
```

Figure 1.24: Structure of the Algorithm. Similar to GSP. The phases where modification is made are shown with an arrow (⟵).

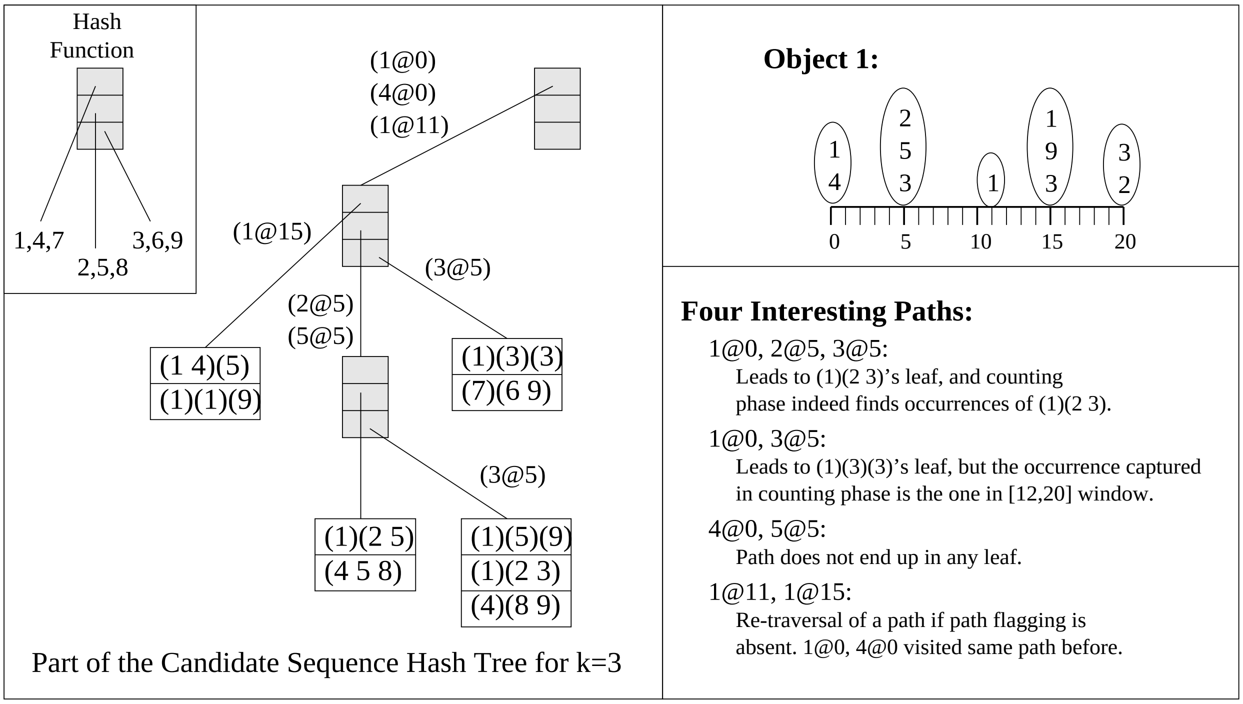


Figure 1.25: Illustrating the candidate hash tree, and the concept of streaming an object's timeline through the hash tree. The significance of using a hash tree can be understood by studying the four interesting paths listed.

set of candidate k-sequences generated by the join-and-prune phase. An efficient way to search occurrences of candidates is by storing them in a hash tree structure similar to the Apriori algorithm for non-sequential associations. An example of a hash tree is shown in Figure 1.25. A candidate is inserted into the hash tree by hashing on the events in that candidate. i^{th} event in the candidate is hashed at i^{th} level of the tree, where root is considered to be at first level. For example, while storing (1)(5)(9), event 1 is hashed at the root node N1, which leads to the leftmost child N2, where event 5 is hashed to reach its middle child N3, and finally event 9 is hashed to reach the rightmost leaf child of node N3. The depth of the hash tree and the number of internal nodes in the hash tree can be controlled by tuning the size of the hash table at internal nodes and the maximum number of candidates allowed to be stored at a leaf node.

The advantage gained out of constructing a hash tree is that the timeline of an object can be streamed through the branches of hash tree to identify only those candidates that could potentially occur in that object's timeline. An example of streaming a timeline is shown in Figure 1.25. The timeline is streamed through the hash tree by assuming that any event in the timeline can be the potential first event of a sequence, hence each event is hashed at the root node. The next event to hash is found using the timing constraints. As an example, let $ms = 10$, $xg = 9$, and $ws = 0$ be the timing constraints. If event 1 occurring at time 0 (abbreviated as 1@0) is hashed at the root, then the only events eligible to be hashed at node N2 are 4@0, 2@5, 3@5, and 5@5. After hashing 5@5 at N2, although the event 1@12 satisfies the xg constraint, it does not satisfy the ms constraint, so it is not hashed at N3. Formally, if an event hashed at the root node occurs at time t_0, and if an event occurring at time t is just hashed at some node, then the event that is eligible to be hashed next must occur in the time-window defined by $[t - ws, \min(t_0 + ms, \max(t + xg, t + ws))]$.

In the process of streaming, when a timeline for an object reaches a leaf node, the algorithm counts all the occurrences of all the candidates stored at that leaf. Remember that we have defined the number of occurrences to be the number of span-size windows in which a sequence occurs. The first occurrence of a sequence is found using the method described in [45]. This search is based on searching event-sets in the order they appear in the sequence. The ws constraint is checked while searching an individual event-set.

If an event-set is not found, search terminates. If event-set is found but it does not satisfy the ng and xg constraints with respect to the previous event-set, then a search is performed in the reverse direction to find occurrences of all previous event-sets within the ng and xg constraints. If the event-set is found, but is over the ms limit, then a new search is started from time $t_e - ms$, where t_e is the time of the latest event occurrence in the recently found event-set.

After finding the first occurrence, instead of stopping the search as done in GSP [45], our algorithm continues to search for the next occurrence. This next search starts at time $t+1$, where t is the time at which the earliest event occurred in previously found sequence. Let the first occurrence of a sequence be found in the range [t_s,t_e]. Then the number of span-windows, that this occurrence contributes to, is equal to $(t_s - (t_e - ms + 1))$. Now, the next sequence search starts at time $t_s + 1$. If the next occurrence is found in time range [$t_s^{'}$,$t_e^{'}$], then the number of span-windows that this occurrence contributes to is equal to $(t_s^{'} - max(t_e^{'} - ms, t_s + 1)]$; i.e. count all the span-windows that start after the time t_s and lie within ms duration of $t_e^{'}$. As an example, consider the object 1's timeline shown in Figure 1.25. The constraint ms is 10. The first occurrence of (1)(2 3) is found in the range [0,5], contributing to all the span-windows that which start at time-points between -4 and 0. The next occurrence of (1) (2 3) is found in the range [11,20]. This contributes to the windows starting at time-points 11 and 12. The next occurrence is found to be in the range [15,20], this accounts for the windows starting at time-points 13, 14, and 15. This discussion is relevant when CWIN method is used for counting. For other methods, appropriate changes need to be made. For example, when CDIST method is used, each event-timestamp pair must be flagged to indicate if it has been used towards counting the occurrence of a given sequence. When CDIST_O method is used, the counting process is similar to CWIN, but instead of counting the windows, just one occurrence is added each time a search succeeds.

Following points should be noted regarding the algorithm discussed above.

- The role of a hash tree to increase the efficiency of search can be fulfilled only if the hash tree is not too deep and/or too large, because in the worst case the amount of work involved in streaming a timeline through the hash tree is equal to the amount of work done in finding first occurrence of each candidate using a straight-forward search (when every candidate is stored at a different leaf in the hash tree). On the other hand, if the number of candidates is very large *and* the tree is very shallow or small, then the advantage of building a hash tree would be lost again. Hence, the size of the hash table at each node and the maximum number of candidates allowed to be stored at leaf nodes play an important role in determining the efficiency of the algorithm.
- If an object's timeline is very large, then the operation of streaming a timeline needs to be optimized to flag the paths which are impossible and the root-to-leaf paths which are already traversed. This avoids their repeated traversal.
- If an object's timeline is very large, then the counting operation at the leaf can possibly take a very long time because the entire timeline would need to be traversed for each candidate at every leaf reached.

1.6.6 Applications of Universal Sequential Patterns

Let us see one representative application, where universal formulation presented in this paper is required and the formulations suggested so far in the literature are not suitable.

- **An Illustrative Application**: Discovering Consumer Buying Patterns.

  Let us consider an example of a grocery store.

  Goal: To find out "which items' sales trigger sales in other items within a period of one week" so that the information can help in managing weekly inventory.

  Formulation:

Object: Customer
Event: Items bought by the customer
Timestamp: Date of transaction

Constraints:
xg = 2 days, ng = 0 days, ws = 0 days, ms = 7 days.

This constraints restrict all the items in an event-set to be bought on the same day (ws=0), the pattern span does not exceed 7 days (ms), and a pattern like (Eggs)(Bread) will be supported by a customer only if he buys Eggs today and Bread tomorrow or day-after-tomorrow (xg=2). Finally, ng=0 means that items A and B bought on the same day cannot occur on two different nodes separated by an edge.

Counting Method:
Let us consider a pattern (Orange Juice)(Mayonnaise)(Frozen Burger).

COBJ: If pattern above is found to be frequent with this method, then it can be concluded that *many customers* who buy Orange Juice today will buy Mayonnaise within a couple days followed by Frozen Burger within a couple of days more, *but* they will buy all these items within a week. The frequent patterns generated with this method will tell *how many customers* are likely to exhibit that pattern. Note that, this method does not try to increase the strength of the pattern if some of the customers exhibited this pattern multiple number of times, which might be important in some cases.

CDI ST: If the pattern above is found to be frequent with this method, then it implies that many distinct occurrences of the pattern were exhibited when they were added up over all customers. If the goal is to increase the sales of Frozen Burgers, then this counting method gives more practical meaning to pattern. This is because distinct occurrences with no overlap mean that we could find sufficiently many distinct sequences of these items without counting any item more than once. So, the count reflects on how many cans of orange juice and how many Mayonnaise and Burger packs were actually sold.

CWIN, CMINWIN, CDIST_O: Although these counting methods can be applied, they may not be valuable from the practical viewpoint, because none of them would reflect on the true sales of the items involved in the pattern because overlap is allowed in all of them. This means that same can of Orange Juice can contribute to many occurrences or many windows.

This example illustrates that not every counting method can be applicable in all the domains. Nature of the application domain and user's knowledge regarding the domain will determine which counting method to use.

Where Does Universal Formulation Help?:
It seems that the generalized sequential patterns formulation of [45] could have helped in this scenario, but that formulation has only one counting method (COBJ), which does not allow to gain more insight into the patterns which are given by the other method (CDIST) of universal formulation.

Moreover, generalized sequential patterns of [45] do not place any limit on the total pattern duration. Hence, if the available data is collected over a large number of weeks, then in order to make generalized sequential patterns applicable, multiple 7-day-long datasets would need to be formed and analyzed separately. Many issues would arise in that case, such as which 7-day intervals to choose, how to combine the patterns generated with each dataset, etc. Also the method of [46] is not applicable. The fact that the method does not allow multiple objects needs to be handled first. This requires extra preprocessing of input dataset, which includes merging of the datasets for all the customers into a single dataset and separating individual customer's timelines by a gap greater than ms value, so that patterns do not span across two different customers. Even if this extra cost of preprocessing is tolerated, the method does not support COBJ or CDIST counting methods. The only counting method it supports is CWIN, which does not seem to have much practical value in this scenario (as explained above). Also, it does not allow to specify the xg and ng constraints, which can be crucial in this application.

Many more applications are possible. We will list a few of them here.

- **Possible Application Areas for Universal Sequential Patterns:**
  1. Discovering sequential relationships between different telecommunication switches and alarms triggering on them.
  2. Analyzing data from scientific experiments conducted over a period of time.
  3. Discovering relationships between stock market events (e.g. fluctuations happening on market indices, individual stock prices).
  4. Analyzing medical records of patients for temporal patterns between diagnosis, treatment, symptoms, and examination results, etc.
  5. Discovering Patterns Among Different Socio-Economic Events.

It should be noted that each application domain will require a careful combination of timing constraints, event constraints, and counting methods, in order to produce meaningful patterns. With universal formulation, one can easily try out multiple such combinations, specify different structures of the relationships, and the same algorithm is applicable in all the cases. This is real strength of the universal sequential patterns. The models that were previously suggested in the literature do not have enough set of constraints for a user to focus on specific kinds of patterns, and they do not support different counting methods. Allowing different counting methods to be supported in one unifying formulation, ultimately allows a user to encode different hypotheses he/she has about the sequential relationships. This can range from hypothesizing nothing but a few timing constraints to putting all possible constraints encoding all the previously known knowledge the user has about the application domain.

1.7 Summary

In this chapter, we presented an evolutionary and comparative review of many existing algorithms for solving a very popular and important problem of mining associations from data. We considered the traditional non-sequential associations which originated from the transaction or market basket kind of data as well as the more generalized sequential association formulation which is useful to wider variety of datasets in real world. The chapter mainly elaborates on various design issues involved in parallel formulations of association discovery algorithms, and how existing parallel algorithms map to only a few categories of formulations. In the process, a comprehensive survey of many existing serial algorithms is also given. Although many parallel (and serial) algorithms exist today, no single algorithm is superior to all the rest, and the research in the discovery of associations remains active. In particular, there are many serial algorithms that have emerged recently, for which no effective parallel algorithms have been developed yet. Even some existing parallel algorithms can be improved for their scalability, portability to different kinds of architectures, or applicability to generalized formulations.

Overall, this chapter provides a comprehensive account of existing serial and parallel algorithms for mining non-sequential as well as sequential associations with respect to the design issues and different parallelization strategies.