10 Mining Episodes

In the episode model, the data is a history of *events*; each event has a *type* and a time of occurrence. An example of event type might be: "switch 34 became overloaded and had to drop a packet." It is probably too general to have an event time of the form "some switch became overloaded." Several events may occur at the same time, and there is no guarantee that some event happens at every time unit.

10.1 Applications of Epsiode Mining

- Mining "epsiodes," that is, sequences or sets of event types that occur within a short window, has been used to help predict outages in the Finnish power grid; the paper of MTV is an abstraction of this work.
- The same ideas could be used to monitor packet-switching or other communication networks, to develop rules for rerouting data in advance of congestion. It may also be possible to mine for rules that predict failures from combinations of manifestations in a variety of complex systems.

10.2 Epsiodes

- A *parallel episode* is a set of event types, e.g., $\{A, B, C\}$. In diagrams, these episodes are represented by a vertical box with A, B, and C within. The intent of a parallel episode is that each of the events in the episode occurs (within a window of time), but the order is not important.
- A *serial episode* is a list of event types, e.g., (A, B, C). In diagrams, these events are shown in a horizontal box, in order. The intent is that within a window of time, these events occur in order. Note that a single event may be thought of as both a parallel and a serial episode.
- A *composite episode* is built recursively from events by serial and parallel composition. That is, a composite episode is either:
  - An event,
  - The serial composition of two or more events, or
  - The parallel composition of two or more events.

  Thus. every serial and parallel episode is also a composite episode, but there are episodes that are composite, yet not serial or parallel.

Example 10.1 : Figure 29 shows a composite episode. It is the serial composition of three episodes. The first is the single event A. Then comes the parallel epsiode $\{B, C, D\}$. The third is a composite episode consisting of the parallel composition of the serial episodes (E, F) and (G, H). Two examples of orders of these 8 events that are consistent with this episode are $ABCDEGFH$ and $ACDBGHEF$. $\Box$

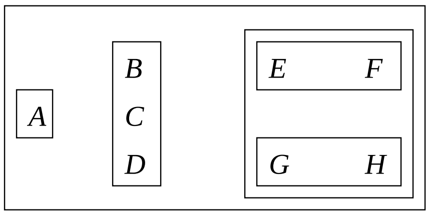

Figure 29: A composite episode

10.3 Monotonicity of Episodes and the A-Priori Algorithm

Given a window length w (an amount of time during which an episode may occur), an episode is *frequent* if it occurs in at least s windows, where s is the support threshold. Note that the same sequence of events may show up as an episode in several consecutive windows; we count one unit for each such window. However, no window can be credited with having the same episode occur more than once, even if we can construct the episode from several different sets of events in the same window.

Like frequent itemsets, frequent episodes are monotone: if an episode E is frequent, then so is any episode formed by deleting some events from E. Thus, we can construct all frequent episodes "levelwise," using a trick that is very much like a-priori.

1. Let C_i be the candidate episodes of size (number of events) i and let L_i be the frequent episodes of size i.

2. For a basis, C_1 is the set of all event types.

3. Construct C_{i+1} from L_i by putting in C_{i+1} exactly those episodes E of size $i+1$ such that deleting any one event from E yields an event from $L+i$.

Example 10.2: The epsiode of size 3 in Fig. 30(a) can be frequent (i.e., in C_3) only if the three episodes of Fig. 30(b) are frequent (i.e., in L_2). □

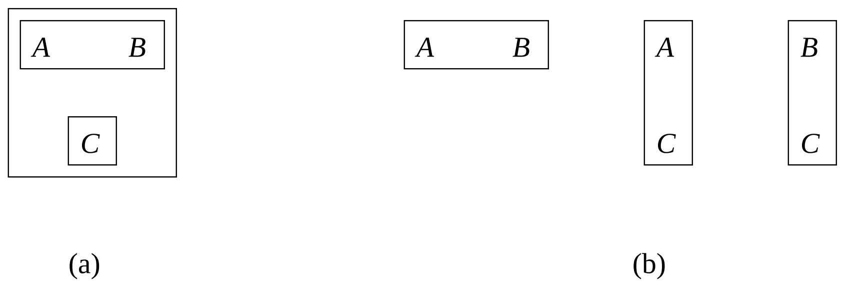

Figure 30: Episode (a) is a candidate in C_3 only if all three episodes (b) are in L_2

10.4 Checking Parallel Episodes

The big problem is converting C_i into L_i by going once through the data and counting the support for each episode in C_i. A dumb algorithm will look at each window of length w in turn, and for each candidate episode check whether it is there; if so, add 1 to the support for the episode.

The goal of MTV is to perform work that is only proportional to the sum over all events of the number of candidate episodes (including the subepisodes of composite episodes) that have that event. The following algorithm was developed in class; it seems less restrictive than the one given in the paper, since it does not assume the impossibility of an event type appearing several times in a window. We describe how the data is scanned once, to compute L_i from C_i, considering each window in turn, from the beginning of the sequence. The data structure needed:

1. For each event type A:

   (a) A count `A.count` of the number of times A has been seen in the present window.

   (b) A linked list `A.contains` of all the events E that contain A.

2. For each candidate episode E:

   (a) A time `E.startingTime` that is the beginning of a consecutive sequence of windows in which E has always been present, up to the present window.

(b) An integer `E.support`, the number of windows in which E has appeared, not including windows since `startingTime`.

(c) An integer `E.missing` giving the number of events A of E that are *not* in the present window. Note that E is present in the window if and only if `E.missing==0`.

The heart of the algorithm is what we do when we slide the window one time unit forward. We must consider what happens when an event A drops out of the beginning, and what happens when an event B is included at the front. If A drops out of the window:

```
     A.count--;
     if(A.count==0) /* we just lost A */
 for(all E on A.contains) {
             E.missing++;
             if(E.missing==1) /* we just lost E */
                 E.support += (E.startingTime - currentTime);
         }
```

If B enters the window:

```
     B.count++;
     if(B.count==1) /* we just gained B */
         for(all E on B.contains) {
             E.missing--;
             if(E.missing==0) /* we just gained E */
                 E.startingTime = currentTime;
         }
```

10.5 Checking Serial Episodes

To check serial episode $(A_1, A_2, \ldots, A_n)$ we simulate a nondeterministic finite automaton that recognizes the string $.* A_1 A_2 \cdots A_n$, as suggested by Fig. 31. As we scan the events in the data, we keep track of the set of states that the NFA is in.

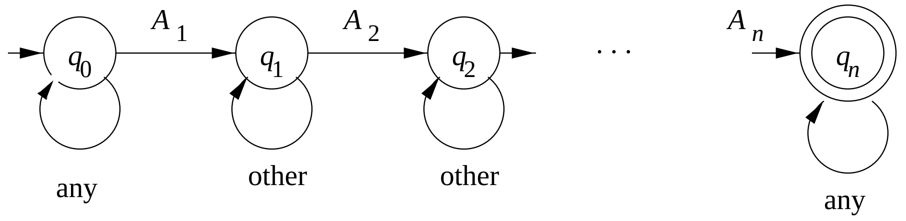

Figure 31: An NFA for recognizing a serial episode

Here is a rough description of how the NFA's for the various events E are used, assuming a data structure similar to that described in Section 10.4 for parallel episodes.

- Note that the NFA stays in the same set of states it was in if the input symbol is an event that is not part of its episode. Thus, simulating this automaton requires 0 time unless its episode is on a list like `A.contains` for the current event A (see the data structure in Section 10.4).
- As we simulate, we keep for each state the NFA is currently in the *most recent* point at which we could have started the NFA and still gotten to that state. This value is:
  1. The current time if we enter state q_1.
  2. The time of q_{i-1} if we enter state q_i by reading A_i.

3. The same time as previously, if we were already in q_i and the next input is other than A_i.

- If the NFA for episode E enters the accepting state q_n, but was not previously in that state, then set `E.startingTime` to the time currently associated with q_n.
- If any time associated with a state is less that the current time minus w (the window length), then delete that state from the set of states the NFA is in. If the accepting state is thus lost, add `E.startingTime` minus the current time to `E.support`.

Example 10.3: Suppose the event E is (A, B, C). The NFA for E is as in Fig. 32.

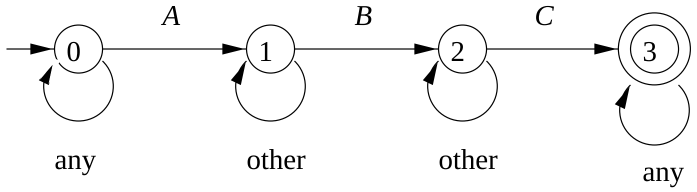

Figure 32: NFA for the episode (A, B, C)

Let the data include the following occurrences of A, B, and C, along with other event types, not shown: (A, B, A, B, C); all these events are within the current window. Then the states of the NFA after each of these events is as shown in Fig. 33. Solid lines show how each state is derived from the previous one by transitions of the NFA, and dashed lines indicate the associated time for each state. □

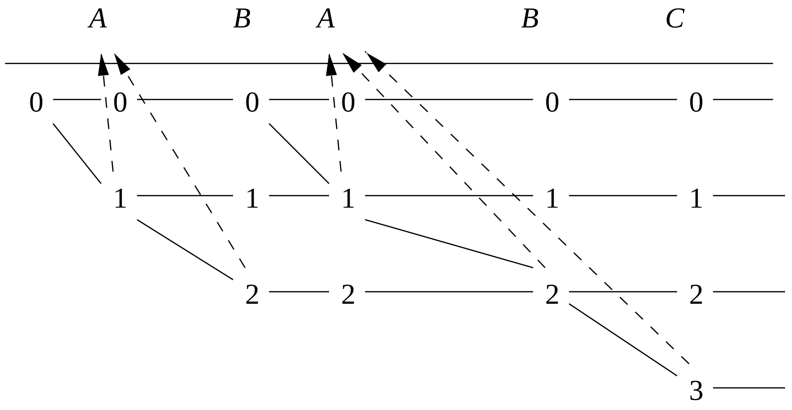

Figure 33: Maintaining the set of states and their associated times

10.6 Counting Composite Events

To extend the above ideas to composite episodes, we must keep a "machine" of some sort for each subepisode. It is the job of each machine to report to the machine(s) for the superepisode(s) of which it is a part whether or not it is present, and if so what is the most recent time at which it can be construed to have begun.

- If the subepisode is the parallel composition of events and/or other subepisodes, we use a data structure like that of Section 10.4, but we replace the count for an event by the most recent beginning time for a composite episode.
- For serial episodes, we maintain an automaton, but the inputs are occurrences of its subepisodes.
- The starting time and support only need to be maintained for episodes in C_i, not for subepisodes.