Finding Frequent Patterns in Parallel Point Processes

Christian Borgelt and David Picado-Muiño

European Centre for Soft Computing
Gonzalo Gutiérrez Quirós s/n, 33600 Mieres, Spain
{christian.borgelt|david.picado}@softcomputing.es

Abstract. We consider the task of finding frequent patterns in parallel point processes—also known as finding frequent parallel episodes in event sequences. This task can be seen as a generalization of frequent item set mining: the co-occurrence of items (or events) in transactions is replaced by their (imprecise) co-occurrence on a continuous (time) scale, meaning that they occur in a limited (time) span from each other. We define the support of an item set in this setting based on a maximum independent set approach allowing for efficient computation. Furthermore, we show how the enumeration and test of candidate sets can be made efficient by properly reducing the event sequences and exploiting perfect extension pruning. Finally, we study how the resulting frequent item sets/event sets can be filtered for closed and maximal sets.

1 Introduction

We present methodology and algorithms to identify *frequent patterns* in parallel point processes, a task that is also known as finding *frequent parallel episodes* in event sequences (see [7]). This task can be seen as a generalization of frequent item set mining (FIM)—see e.g. [2]. While in FIM items co-occur if they are contained in the same transaction, in our setting a continuous (time) scale underlies the data and items (or events) co-occur if they occur in a (user-defined) limited (time) span from each other. The main problem of this task is that, due to the absence of (natural) transactions, counting the number of co-occurrences (and thus determining what is known as the *support* of an item set in FIM) is not a trivial problem. In this paper we rely on a *maximum independent set* approach, which has the advantage that it renders support *anti-monotone*. This property is decisive for an efficient search for frequent patterns, because it entails the so-called *apriori property*, which allows to prune the search effectively. Although NP-complete in the general case, the maximum independent set problem can be solved efficiently in our case due to the restriction of the problem instances by the underlying one-dimensional domain (i.e., the continuous time scale).

The application domain that motivates our investigation is the analysis of *parallel spike trains* in neurobiology: sequences of points in time, one per neuron, representing the times at which an electrical impulse (*action potential* or *spike*)

is emitted. Our objective is to identify *neuronal assemblies*, intuitively understood as groups of neurons that tend to exhibit synchronous spiking. Such cell assemblies were proposed in [5] as a model for encoding and processing information in biological neural networks. In particular, as a (possibly) first step in the identification of neuronal assemblies, we look for *frequent neuronal patterns* (i.e., groups of neurons that exhibit *frequent synchronous spiking*).

The remainder of this paper is structured as follows: Section 2 covers basic terminology and notation. In Section 3 we compare two characterizations of synchrony: *bin-based* and *continuous*, exposing the challenges presented by them. In Section 4 we present our maximum independent set approach to support counting as well as an efficient algorithm. In Section 5 we employ an enumeration scheme (directly inspired by common FIM approaches) to find all frequent patterns in a set of parallel point processes. In particular, we introduce core techniques that are needed to make the search efficient. In Section 6 we present experimental results, demonstrating the efficiency of our algorithm scheme. Finally, in Section 7 we draw conclusions from our discussion.

2 Event Sequences and Parallel Episodes

We (partially) adopt notation and terminology from [7]. Our data are (finite) *sequences of events* of the form $\mathcal{S} = \{\langle e_1, t_1\rangle, \ldots, \langle e_k, t_k\rangle\}$, for $k \in \mathbb{N}$, where e_i in the *event* $\langle e_i, t_i\rangle$ is the *event type* (taken from a domain set E) and $t_i \in \mathbb{R}$ is the time of occurrence of e_i, for all $i \in \{1, ..., k\}$. We assume that $\mathcal{S}$ is ordered with respect to time, that is, $\forall i \in \{1, ..., k-1\}\colon t_i \leq t_{i+1}$. Such data may be represented as *parallel point processes* $\mathcal{L} = \{(a_1, [t_1^{(1)}, \ldots, t_{k_1}^{(1)}]), \ldots, (a_m, [t_1^{(m)}, \ldots, t_{k_m}^{(m)}])\}$ by grouping events with the same type $a_i \in E$ and listing the times of their occurrences (also sorted with respect to time) for each of them.[1] We employ both representations, based on convenience. In our motivating application (i.e., spike train analysis), the event types are given by the neurons and the corresponding point processes list the times at which spikes were recorded for these neurons.

Episodes (in $\mathcal{S}$) are defined as sets of event types in E embedded with a partial order, usually required to occur in $\mathcal{S}$ within a certain time span. *Parallel episodes* have no constraints on the relative order of their elements. An *instance (or occurrence) of a parallel episode* (or a *set of synchronous events*) $A \subseteq E$ in $\mathcal{S}$ with respect to a (user-specified) time span $w \in \mathbb{R}^+$ can be defined as a subsequence $\mathcal{R} \subseteq \mathcal{S}$, which contains exactly one event per event type $a \in A$ and in which any two events are separated by a time distance at most w.

Event synchrony is formally characterized by means of the operator $\sigma_{\mathcal{C}}$:

$$\sigma_{\mathcal{C}}(\mathcal{R}, w) = \begin{cases} 1 & \text{if } \max\{|t_i - t_j| \mid \langle e_i, t_i\rangle, \langle e_j, t_j\rangle \in \mathcal{R}\} \leq w, \\ 0 & \text{otherwise.} \end{cases}$$

In words, $\sigma_{\mathcal{C}}(\mathcal{R}, w) = 1$ if all events in $\mathcal{R}$ lie within a distance at most w from each other; otherwise $\sigma_{\mathcal{C}}(\mathcal{R}) = 0$. The set of all instances of parallel episodes

[1] We use square brackets (i.e., [...]) to denote *lists*.

(or all sets of synchronous events) of a set $A \subseteq E$ of event types is denoted by $\mathcal{E}(A, w)$, which we formally define as

$$\mathcal{E}(A, w) = \big\{\mathcal{R} \subseteq \mathcal{S} \mid A = \{e \mid \langle e, t\rangle \in \mathcal{R}\} \wedge |\mathcal{R}| = |A| \wedge \sigma_{\mathcal{C}}(\mathcal{R}, w) = 1\big\}.$$

That is, $\mathcal{E}(A, w)$ contains all event subsets of $\mathcal{S}$ with exactly one event for each event type in A that lie within a maximum distance of w from each other.

3 Event Synchrony

Although the above definitions are clear enough, the intuitive notion of the (total) amount of synchrony of a set A of event types suggested by it poses problems: simply counting the occurrences of parallel episodes of A—that is, defining the *support* of A (i.e. the total synchrony of A) as $s(\mathcal{E}(A, w)) = |\mathcal{E}(A, w)|$—has undesirable properties. The most prominent of these is that the support/total synchrony of a set $B \supset A$ may be larger than that of A (namely if an instance of a parallel episode for A can be combined with multiple instances of a parallel episode for $B \setminus A$), thus rendering the corresponding support measure not *anti-monotone*. That is, such a support measure does not satisfy $\forall B \supset A : s(\mathcal{E}(B, w)) \leq s(\mathcal{E}(A, w))$. However, this property is decisive for an efficient search for frequent parallel episodes, because it entails the so-called *apriori property*: given a user-specified minimum support threshold $s_{\min}$, we have $\forall B \supset A : s(\mathcal{E}(A, w)) \leq s_{\min} \Rightarrow s(\mathcal{E}(B, w)) \leq s_{\min}$ (that is, *no superset of an infrequent set is frequent*). This property allows to prune the search effectively: as soon as an infrequent set is encountered, no supersets need to be considered anymore.

In order to overcome this problem, many approaches to find frequent parallel episodes resort to *time binning* (including [7] and virtually all approaches employed in neurobiology): the time interval covered by the events under consideration is divided into (usually disjoint) time bins of equal length (i.e., the originally continuous time scale is *discretized*). In this way transactions of classical frequent item set mining (FIM) [2] are formed: events that occur in the same time bin are collected in a transaction and are thus seen as synchronous, while events that occur in different time bins are seen as not synchronous. Technically, time binning can be characterized by the synchrony operator $\sigma_{\mathcal{B}}$, defined as follows for $\mathcal{R} \subseteq \mathcal{S}$ and w now representing the bin width:

$$\sigma_{\mathcal{B}}(\mathcal{R}, w) = \begin{cases} 1 & \text{if } \exists k \in \mathbb{Z}\colon\ \forall \langle e, t\rangle \in \mathcal{R}\colon\ t \in (w(k-1), wk], \\ 0 & \text{otherwise} \end{cases}$$

(implicitly assuming that binning is anchored at 0). Clearly, this solves the problem pointed out above: counting the bins in which all event types of a given set A occur (or, equivalently, counting the number of sets of synchronous events of A—at most one per time bin) yields an anti-monotone support measure. However, this *bin-based* model of synchrony has several disadvantages:

– **Boundary problem**. Two events separated by a time distance (much) smaller than the bin length may end up in different bins and thus be regarded as non-synchronous. Such behavior is certainly undesirable.

- **Bivalence problem**. Two events can be either (fully) synchronous or non-synchronous. Small variations in the time distance between two spikes (possibly moving one of them over a bin boundary) cause a jump from (full) synchrony to non-synchrony and *vice versa*. This may be counter-intuitive.
- **Clipping**. In neurobiology the term *clipping* refers to the fact that in a bin-based model it is usually considered only *whether* a neuron emits a spike in a time bin, not *how many* spikes it emits in it. The same can be observed for general event types in many settings employing time binning.

Some of these disadvantages can be mitigated by using overlapping time bins, but this causes other problems, especially certain anomalies in the counting of parallel episodes if they span intervals of (widely) different lengths: parallel episodes with a short span may be counted more often than parallel episodes with a long span, because they can occur in more (overlapping) time bins.

Due to these problems we prefer a synchrony model that does *not* discretize time into bins, but rather keeps the (time) scale continuous: a *continuous* synchrony model, as it was formalized in Section 2. This model captures the *intended* characterization of synchrony in the bin-based approach, solves the boundary problem and overcomes the effects of clipping, while it keeps the synchrony notion bivalent (for a related continuous model with a graded notion of synchrony see [8]). In order to overcome the anti-monotonicity problem pointed out above, we define the support of a set $A \subseteq E$ of event types as follows (see also [6] for a similar characterization):

$$s(\mathcal{E}(A,w)) = \max\big\{|H| \mid H \subseteq \mathcal{E}(A,w) \wedge \nexists \mathcal{R}_1, \mathcal{R}_2 \in H : \mathcal{R}_1 \neq \mathcal{R}_2 \wedge \mathcal{R}_1 \cap \mathcal{R}_2 \neq \emptyset\big\}.$$

That is, we define the support (or total synchrony) of a pattern $A \subseteq E$ as the size of a maximum independent set of instances of parallel episodes of A (where by *independent set* we mean a collection of instances that do not share any events, that is, the instances do not overlap). Such an approach has the advantage that the resulting support measure is guaranteed to be anti-monotone, as can be shown generally for maximum independent subset (or, in a graph interpretation, node set) approaches—see, e.g., [3] or [10].

4 Support Computation

A core problem of the support measure defined in the preceding section is that the maximum independent set problem is, in the general case, NP-complete and thus not efficiently solvable (unless $\mathcal{P} = \mathcal{NP}$). However, we are in a special case here, because the domain of the elements of the sets is one-dimensional and the elements of the considered sets are no more than a (user-specified) maximum distance w apart from each other. The resulting constraints of the possible problem instances allow for an efficient solution, as shown in [9].

Intuitively, the constraints allow to show that a maximum independent set can be found by a greedy algorithm that always selects the next (with respect to time) selectable instance of the parallel episode we are considering. The idea of

```
type train = (int, list of real);          (* pair of identifier and list of points/times *)
                                           (* points/times are assumed to be sorted *)

function support (L: set of train, w: real) : int;
begin                                      (* L: trains to process, w: window width *)
  s := 0;                                  (* initialize the support counter *)
  while ∀(i,l) ∈ L: l ≠ [] do begin        (* while none of the lists is empty *)
    t_min = min {head(l) | (i,l) ∈ L};     (* get smallest and largest head element *)
    t_max = max{head(l) | (i,l) ∈ L};      (* and thus the span of the head elements *)
    if t_max − t_min > w                   (* if not synchronous, delete smallest heads *)
    then L := {(i, tail(l)) | (i,l) ∈ L ∧ head(l) = t_min}
            ∪ {(i,l)        | (i,l) ∈ L ∧ head(l) ≠ t_min};
    else  L := {(i, tail(l)) | (i,l) ∈ L}; (* if synchronous, delete all heads *)
          s := s + 1; end                  (* (i.e., delete found synchronous points) *)
  end                                      (* and increment the support counter *)
  return s;                                (* return the computed support *)
end    (* support() *)
```

Fig. 1. Pseudo-code of the support computation from a set of trains/point processes.

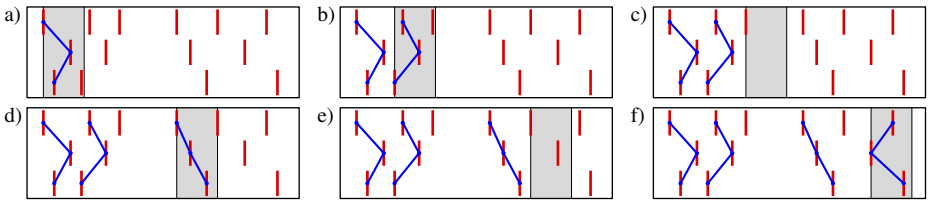

Fig. 2. Illustration of the support computation with a sliding window for three parallel point processes. Blue lines connect selected (i.e. counted) groups of points/times.

the proof is that starting from an arbitrary maximum independent set, the selection of the sets can be modified (while keeping the number of sets and thus the maximality property of the selection), so that events occurring at earlier times are chosen. In the end, the first selected set contains the earliest events of each type that together form an instance of the parallel episode A under consideration. The second selected set contains the earliest events of each type that form an instance of the parallel episode A in an event sequence from which the events of the first instance have been removed, and so on. As a consequence, always selecting greedily the next instance of the parallel episode in time that does not contain events from an already selected one yields a maximum independent set. The details of the proof can be found in [9].

Pseudo-code of the resulting greedy algorithm to compute $s(\mathcal{E}(A, w))$, which works on a representation of the data as *parallel trains* (or parallel point processes) is shown in Figure 1. An illustration in terms of a sliding window that stops at certain points, namely always the next point that is not yet part of a selected set or has already been considered, is shown in Figure 2.

```
function isect (I: list of interval, l: list of real, w: real) : list of interval;
begin                                          (* — intersect interval and point list *)
  J := []; p := −∞; q := −∞;                    (* init. result list and output interval *)
  while I ≠ [] ∧ x ≠ [] do begin               (* while both lists are not empty *)
    a, b := head(I); t := head(l);             (* get next interval and next point *)
    if    t < a then l := tail(l);             (* point before interval: skip point *)
    elif  t > b then I := tail(I);             (* point after  interval: skip interval *)
    else x := max{a, t − w};                   (* if current point is in current interval, *)
         y := min {b, t + w};                  (* intersect with interval around point *)
         if x ≤ q then q := y;                 (* merge with output interval if possible *)
         else if q > −∞ then J.append([p, q]); p := x; q := y; end
         l := tail(l);                         (* store pending output interval, *)
    end                                        (* start a new output interval, and *)
  end                                          (* finally skip the processed point t *)
  if q > −∞ then J.append([p, q]); end         (* append the last output interval *)
  return J;                                    (* return the created interval list *)
end    (* isect() *)

function recurse (C, L: set of train, I: list of interval, w: real, s_min: int);
begin                                          (* — recursive part of CoCoNAD *)
  while L ≠ ∅ begin                            (* while there are more extensions *)
    choose (i, l) ∈ L; L := L − {(i, l)};      (* get and remove the next extension *)
    J := isect(I, l);                          (* intersect interval list with extension *)
    D := {(i′, [t | t ∈ l′ ∧ ∃j ∈ J : t ∈ j]) | (i′, l′) ∈ C ∪ {(i, l)}};
    s := support(D, w);                        (* filter collected trains with intervals, *)
    if s < s_min then continue;                (* compute support of the train set and *)
    report {i′ | (i′, l′) ∈ D} with support s; (* skip infrequent/report frequent sets *)
    X := {(i′, [t | t ∈ l′ ∧ ∃j ∈ J : t ∈ j]) | (i′, l′) ∈ L};
    recurse(D, X, J, w, s_min);                (* filter extensions with interval list and *)
  end                                          (* find frequent patterns recursively *)
end    (* recurse() *)

function coconad (L: set of train, w: real, s_min: int);
begin                                          (* L: list of trains to process *)
  recurse([], L, [[−∞, +∞]], w, s_min);        (* w: window width, s_min: min. support *)
end    (* coconad() *)                        (* initial interval list is whole real line *)
```

Fig. 3. Pseudo-code of the recursive enumeration (support computation see Fig. 1).

5 Finding Frequent Patterns

In order to find all frequent patterns we rely on an enumeration scheme that is directly inspired by analogous approaches in FIM, especially the well-known *Eclat algorithm* [11]. This algorithm uses a *divide-and-conquer* scheme, which can also be seen as a depth-first search in a tree that results from selecting edges of the Hasse diagram of the partially ordered set $(2^E, \subseteq)$—see, e.g., [2]. For a chosen event type a, the problem of finding all frequent patterns is split into two

subproblems: (1) find all frequent patterns containing a and (2) find all frequent patterns *not* containing a. Each subproblem is then further divided based on another event type b: find all frequent patterns containing (1.1) both a and b, (1.2) a but not b, (2.1) b but not a, (2.2) neither a nor b etc. More details of this approach in the context of FIM can be found in [2]. Pseudo-code of this scheme for finding frequent patterns is shown in Figure 3, particularly in the function "recurse": the recursion captures including another event type (first subproblem), the loop excluding it afterwards (second subproblem). Note how the *apriori property* (see above) is used to prune the search.

A core difference to well-known FIM approaches is that we cannot, for example, simply intersect lists of transaction identifiers (as it is done in the Eclat algorithm, see [11] or [2]), because the continuous domain requires keeping all trains (i.e. point processes) of the collected event types in order to be able to compute the support with the function shown in Figure 1. However, simply collecting and evaluating complete trains leads to considerable overhead that renders the search unpleasantly slow (see the experimental results in the next section). In order to speed up the process, we employ a filtering technique based on a list of (time) intervals in which the points/times of the trains have to lie to be able to contribute to instances of a parallel episode under consideration (and its supersets). The core idea is that a point/time in a train that does not have a partner point/time in all other trains already collected (in order to form an instance of a parallel episode) can never contribute to the support of a parallel episode (or any of its supersets) and thus can be removed.

The initial interval list contains only one interval that spans the whole real line (see the main function "coconad" in Figure 3). With each extension of the current set of event types (that is, each split event type in the divide-and-conquer scheme outlined above), the interval list is "intersected" with the train (that is, its list of points/times). Pseudo-code of this intersection is shown in the function "isect" in Figure 3: only spikes lying inside an existing interval are considered. In addition, the intervals are intersected with the intervals $[t - w, t + w]$ around the point/time t in the train, where w is the (user-specified) window width that defines the maximum time distance between events that are to be considered synchronous. The resulting intersections are then merged into a new interval list.

These interval lists are used to filter both the already collected trains before the support is determined (cf. the computation of the set D in the function "recurse" in Figure 3) as well as the potential extensions of the current set of event types (cf. the computation of the set X). Preliminary experiments that we conducted during the development of the algorithm showed that each of these filtering steps actually improves performance (considerably).

A common technique to speed up FIM is so-called *perfect extension pruning*, where an item i is called a *perfect extension* of an item set I if I and $I \cup \{i\}$ have the same support. The core idea is that a subproblem split (as described in the divide-and-conquer scheme above) can be avoided if the chosen split item is a perfect extension. The reason is that in this case the solution of the first subproblem (include the split item) can be constructed easily from the solution of the second (exclude the split item): simply add the perfect extension item

to all frequent item sets in the solution of the second subproblem (see [2] for details). However, for this to be possible, it is necessary that the property of being a perfect extension carries over to supersets, that is, if an item i is a perfect extension of an item set I, then it is also a perfect extension of all item sets $J \supset I$. Unfortunately, this does not hold in the continuous case we consider here: there can be patterns $A, B \subseteq E$, with $B \supset A$, and $a \in E$ such that $s(\mathcal{E}(A, w)) = s(\mathcal{E}(A \cup \{a\}, w))$, but $s(\mathcal{E}(B, w)) > s(\mathcal{E}(B \cup \{a\}, w))$. As a consequence, perfect extension pruning cannot be applied directly.

Fortunately, though, we are still able to employ a modified version, which exploits the fact that we can choose the order of the split event types independently in different branches of the search tree. The idea is this: if we find a perfect extension (based on the support criterion mentioned above), we collect it and only solve the second subproblem (exclude the split event type). Whenever we report a pattern A as frequent we check whether this set together with the set B of collected perfect extensions has the same support. If it does, the property of being a perfect extension actually carries over to supersets in this case. Therefore we can proceed as in FIM: we report all sets $A \cup C$ with $C \subseteq B$, using the same support $s(\mathcal{E}(A, w))$. If not, we "restart" the search using the collected perfect extensions as split/extension items again. Note that due to the fact that we know the support of the set $A \cup B$ (computed to check whether A and $A \cup B$ have the same support), we have an additional pruning possibility: as soon as we reach this support in the restarted recursion, the remaining perfect extensions can be treated like "true" perfect extensions. (Note that this technique is not captured in the pseudo-code in Figure 3; for details refer to the source code, which we made available on the internet—see below).

Finally, we consider filtering for closed and maximal frequent patterns, which is a common technique in FIM to reduce the output size. For applications in spike train analysis we are particularly interested in closed frequent patterns, because they capture all frequency information without loss, but (usually) lead to (much) smaller output (while maximal frequent patterns, which can reduce the output even more, lose frequency information and cause certain unpleasant interpretation problems). Because of this filtering for closed sets of neurons that frequently fire together as an indication of assembly activity, we call our algorithm CoCoNAD (for COntinuous-time ClOsed Neuron Assembly Detection—see Figure 3, although the pseudo-code does not capture this filtering).

The main problem of filtering for closed and maximal sets are the "eliminated event types," that is, event types which have been used as split event types and w.r.t. which we are in the second subproblem (exclude item). While event types that have *not* been used on the current path in the search tree are processed in the recursion and hence it can be returned from the recursion whether there exists a superset containing them that has the same support (closed sets) or is frequent (maximal sets), eliminated event types need special treatment. We implemented two approaches: (1) collecting the (filtered trains of the) eliminated event types and explicitly checking the support of patterns that result from adding them and (2) using (conditional) repositories of (closed) frequent sets as suggested for FIM in [4]. (Again these techniques are not captured in Figure 3;

details can be found in the source code). The latter has the disadvantage that it requires extra memory for the (conditional) repositories, but turns out to be significantly faster than the former (see next section).

6 Results

We implemented our algorithm scheme in both Python and C (see below for the sources) and tested it on the task of identifying all frequent patterns in data sets with varying parameters in order to assess its efficiency. Parameters were chosen with a view on our application domain: data sets with a varying number of event types (i.e. neurons in our application domain), chosen based on the number of neurons that can be simultaneously recorded with current technology (around 100, cf. [1]). Event rates were chosen according to typical firing rates observed in spike train recordings (around 20–30Hz), which usually comprise a few seconds. Several minimum support thresholds and window widths were considered as an illustration. Window widths were selected based on typical time bin lengths in applications of the bin-based model of synchrony (1 to 7 milliseconds). Support thresholds were considered down to two sets of synchronous events to demonstrate that highly sensitive detections are possible.

The first three diagrams in Figure 4 show execution times of our algorithm on some of these data sets to give an impression of what impact the parameters

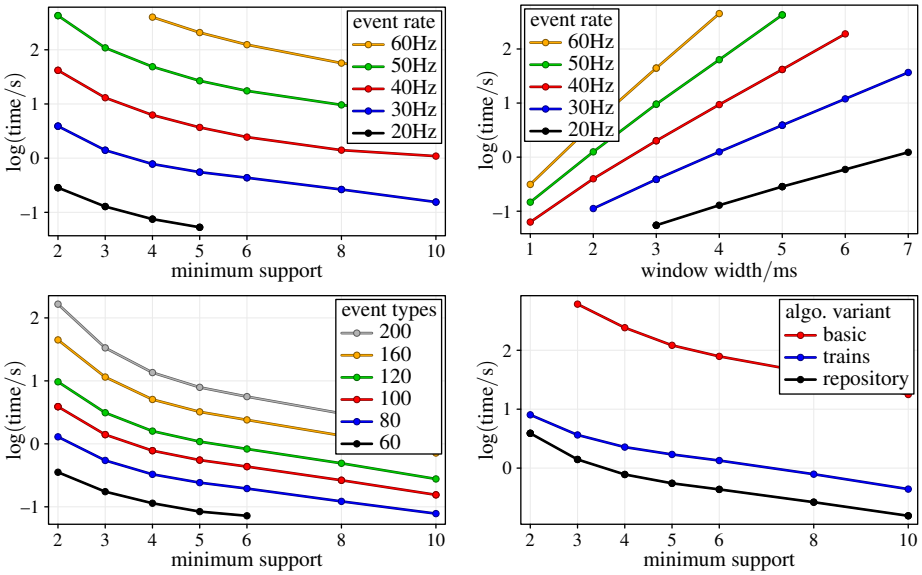

Fig. 4. Execution times in different experimental settings. Default parameters (unless on horizontal axis or in legend) are 5s recording period, 100 event types, 30Hz event rate, $s_{\min} = 2$, $w = 5$ms, closed item sets filtered with (conditional) repositories.

have on the execution time.[2] The last diagram compares the performances of the algorithmic variants described in Section 5. In this diagram "basic" refers to an algorithm without filtering of point processes, that is, as if the pseudo-code in Figure 3 (cf. function "recurse") used the assignments $D := C \cup \{(i, l)\}$ and $X := L$, that is, as if the complete trains were maintained; "trains" means an algorithm with filtering of point processes, where the trains of eliminated items are collected and used to check at reporting time for closed patterns; while "repository" means an algorithm with filtering of point processes, but using (conditional) repositories of already found closed patterns to filter for additional closed patterns. Note that filtering the point processes contributes most to make the search efficient (it reduces the time by about two orders of magnitude).

7 Conclusion

We presented an efficient algorithm scheme aimed at identifying frequent patterns in parallel point processes. This task can be seen as a generalization of frequent item set mining to a continuous (time) scale, where items or events co-occur (that is, are synchronous and thus constitute a set of synchronous events) if they all lie within a certain user-defined (time) span from each other. The main problem of this task is that, due to the absence of natural transactions (on which standard frequent item set mining is based), counting the number of sets of synchronous events (i.e., assessing the support) of a pattern is not a trivial matter. In this paper the support of a pattern is defined as the maximum number of non-overlapping sets of synchronous events that can be identified for that particular set. This has the advantage that it renders support anti-monotone and thus allows to prune the search for frequent patterns effectively. Computing the support thus defined becomes an instance of the maximum independent set problem that, although NP-complete in the general case, can be shown to be efficiently solvable in our case due to the restriction of the problem instances by the underlying one-dimensional domain (i.e., the continuous time scale).

In order to make the search for frequent patterns efficient we introduced several core techniques, such as filtering the point processes to reduce them to the relevant points and using (conditional) repositories to filter for closed (or maximal) patterns. These techniques contribute substantially to speeding up the search, as is demonstrated by the experiments reported in this paper.

Software and Source Code

Python and C implementations of the described algorithm as command line programs as well as a Python extension module that makes the C implementation accessible in Python (2.7.x as well as 3.x) can be found at these URLs:

`www.borgelt.net/coconad.html` `www.borgelt.net/pycoco.html`

[2] All tests were run on a standard PC with an Intel Core 2 Quad 9650@3GHz processor, 8GB RAM, Ubuntu Linux 12.10 64bit operating system, using the C implementation of our algorithm compiled with GCC 4.7.2.

Acknowledgments. The work presented in this paper was partially supported by the Spanish Ministry for Economy and Competitiveness (MINECO Grant TIN2012-31372).

References

1. R. Bhandari, S. Negi, and F. Solzbacher. Wafer Scale Fabrication of Penetrating Neural Electrode Arrays. *Biomedical Microdevices* 12(5):797–807. Springer, New York, NY, USA 2010
2. C. Borgelt. Frequent Item Set Mining. *Wiley Interdisciplinary Reviews (WIREs): Data Mining and Knowledge Discovery* 2:437–456 (doi:10.1002/widm.1074). J. Wiley & Sons, Chichester, United Kingdom 2012
3. M. Fiedler and C. Borgelt. Subgraph Support in a Single Graph. *Proc. IEEE Int. Workshop on Mining Graphs and Complex Data*, 399–404. IEEE Press, Piscataway, NJ, USA 2007
4. G. Grahne and J. Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets. *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003
5. D. Hebb. *The Organization of Behavior*. J. Wiley & Sons, New York, NY, USA 1949
6. S. Laxman, P.S. Sastry, and K. Unnikrishnan. Discovering Frequent Episodes and Learning Hidden Markov Models: A Formal Connection. *IEEE Trans. on Knowledge and Data Engineering* 17(11):1505–1517. IEEE Press, Piscataway, NJ, USA 2005
7. H. Mannila, H. Toivonen, and A. Verkamo. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery* 1(3):259–289. Springer, New York, NY, USA 1997
8. David Picado-Muiño, Iván Castro-León, and Christian Borgelt. Fuzzy Characterization of Spike Synchrony in Parallel Spike Trains. *Soft Computing*, to appear. Springer-Verlag, Berlin, Germany 2013
9. D. Picado-Muino and C. Borgelt. Frequent Itemset Mining for Sequential Data: Synchrony in Neuronal Spike Trains. *Intelligent Data Analysis*, to appear. IOS Press, Amsterdam, Netherlands 2013
10. N. Vanetik, E. Gudes, and S.E. Shimony. Computing Frequent Graph Patterns from Semistructured Data. *Proc. IEEE Int. Conf. on Data Mining*, 458–465. IEEE Press, Piscataway, NJ, USA 2002
11. M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD 1997, Newport Beach, CA)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997