TSRuleGrowth: Mining Partially-Ordered Prediction Rules From a Time Series of Discrete Elements, Application to a Context of Ambient Intelligence

Benoit Vuillemin[1,2(✉)], Lionel Delphin-Poulat[1], Rozenn Nicol[1], Laetitia Matignon[2], and Salima Hassas[2]

[1] Orange Labs, Lannion, France
benoit.vuil@gmail.com
[2] Univ Lyon, Université Lyon 1, CNRS, LIRIS, UMR5205, 69622 Lyon, France

Abstract. This paper presents TSRuleGrowth, an algorithm for mining partially-ordered rules on a time series. TSRuleGrowth takes principles from the state of the art of transactional rule mining, and applies them to time series. It proposes a new definition of the support, which overcomes the limitations of previous definitions. Experiments on two databases of real data coming from connected environments show that this algorithm extracts relevant usual situations and outperforms the state of the art.

Keywords: Rule mining · Ambient intelligence · Habits · Automation · Support · Time series

1 Introduction

Searching for prediction rules in a time series is a major problem in data mining. Used in stock price analysis and recommendation of items for consumers among other fields, this problem has been studied increasingly as the field of ambient intelligence (AmI) expands. AmI is the fusion between artificial intelligence and the Internet of Things, and can be described as: "A digital environment that proactively, but sensibly, supports people in their daily lives" [2]. This work falls within the field of AmI: we want to make a system that finds the habits of users in a connected environment, i.e. an environment in which connected objects are present, in order to provide users with automation.

This paper describes TSRuleGrowth, a new algorithm used in our AmI system, that searches for prediction rules over a time series. Here, this time series represents events sent by connected objects. These prediction rules will then be proposed to users as automation possibilities. TSRuleGrowth uses the principles of a rule mining algorithm on transactions, TRuleGrowth, while adapting them to time series. Also, a new definition of support on time series is described, which overcomes the limitations of the state of the art. In the scope of this paper, the time series is composed only of categorical values, rather than continuous, that can occur at any time, i.e. there is no fixed sampling frequency in the time series. The structure of these rules is described, as well as the state of the art of the fields concerned, which will explain the choices made for this algorithm.

2 Context and Definitions

2.1 Input of Our Ambient Intelligence System

There are two types of connected objects: sensors and actuators. **Sensors** monitor environmental variables. A sensor returns events, corresponding to changes in the state of the observed variable. For example, for a door opening sensor, opening and closing events are sent over time. Sensors can measure **categorical** or **continuous** variables. For example, the temperature of a room, expressed in degrees, can be considered as a continuous variable, while the selection of a radio station, or the opening of a door are categorical variables. A discretization process can convert continuous data into categorical data. **In this paper, only sensors that monitor categorical variables are considered. Actuators** act on the environment. An actuator returns an event when it has made an action. For example, a connected shutter will return an event when it closes or opens. Actuators can perform categorical actions, such as opening a shutter, or continuous actions, like increasing the temperature to a certain value. **As with sensors, only actuators that make categorical actions are considered.**

Each object, whether it is a sensor or an actuator, sends elementary events. In this paper, they are referred to as **elements**. All the elements sent by all the objects are gathered in a set noted E. Let us take the example of a room containing two connected objects: a presence sensor, used to define whether a person is in the room or not, and an actuator: a radio. The presence sensor can detect the following: “Present” and “Absent”. The radio can act in two ways: its power status can be: “Radio on” or “Radio off”, and it can select one of the following stations: “Music”, “News”, “Talk”. Therefore, the set of all elements is $E = \{\text{Present, Absent, Radio on, Radio off, Music, News, Talk}\}$.

Our proposed AmI system collects data streams from several connected objects. Each data stream is composed of a succession of elements, each of which may occur once or several times. Each occurrence is time stamped. Thus, each element is potentially associated with several time data corresponding to its multiple occurrences. For further processing, all collected data from the various objects are aggregated into one single **time series**. In other words, a time series is obtained by a time-ordered concatenation of elements provided by all the individual objects. It is noted $TS{=}\langle(t_1, I_1), ..., (t_n, I_n)\rangle, I_1, ..., I_n \subseteq E$, where:

- t_i is a **time stamp**. It defines a fixed point in time.
- $I_i \subseteq E$ is called an **itemset**. It is the set of individual elements of E which are observed at time stamp t_i

Please note that a given element can only be seen once in an itemset. Also, time stamps are not necessarily equally distributed. The Fig. 1 is an example of a time series created from the environment mentioned in Sect. 2.1. Its mathematical representation is: $TS = \langle$(10:00 am, {Present}), (10:44 am, {Radio on, Music}), (11:36am, {Radio off}), (12:11 am, {Absent}), (2:14 pm, {Present}), (2:52 pm, {Radio on, News}), (3:49 pm, {Music}), (5:14 pm, {Radio off}), (5:57 pm, {Absent})$\rangle$. TS represents activities of a user in the environment. The following section details what the system must find from a time series.

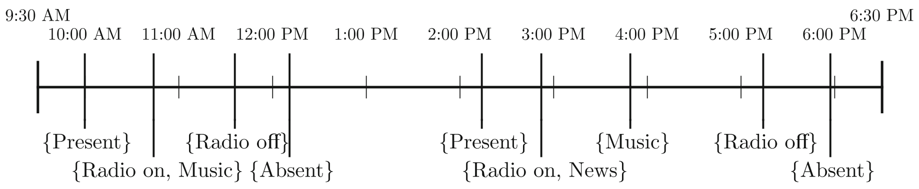

Fig. 1. Representation of a time series

2.2 Output of Our Ambient Intelligence System

The proposed system needs to find **prediction rules**, to express the user's observed habits. A prediction rule is noted $R : E_c \Rightarrow E_p$, where $E_c \subseteq E$ is the **condition**, and $E_p \subseteq E$ is the **prediction** of the rule. R states that if E_c is observed, E_p will be observed after a certain time. A rule must be frequent and reliable to be validated. One could also search for rules in the context of anomaly detection, i.e. very infrequent but highly interesting rules, but this does not fall within the scope of this paper. In the proposed use case, we want to limit the search for rules **for which the prediction part E_p must only be composed of elements originating from actuators**. Indeed, the rule search process being highly combinatorial, this makes it possible to limit this aspect while being adapted to the use case: the system's goal is to propose automatic actions according to situations. According to the time series example in Sect. 2.1, a rule can be {Present} $\Rightarrow$ {Radio on}. This is a **basic rule**, where condition and prediction are composed of a single element. It should be noted that we do not want, for example, to find the {Radio off} $\Rightarrow$ {Absent} rule because its prediction part, {Absent}, comes from a sensor (the presence sensor) and not from actuators (the radio). Several types of prediction rules are possible [7]:

– **Fully-ordered sequential rules**, where the condition E_c and the prediction E_p are sequences, i.e. time ordered successions of elements
– **Partially-ordered sequential rules** [8], where E_c and E_p are both unordered, but an order still exists as E_p comes after E_c. Two mathematical structures are possible for E_c and E_p: **sets**, where an element can only appear once, and **multisets**, where multiple instances of elements are allowed. The number of instances of an element in the multiset is called the **multiplicity**. For example, the multiplicity of the element x in the multiset $\{x, x, y\}$ is 2.

After experimenting with each of them, we chose to use **partially-ordered sequential rules containing multisets**. The problem with fully-ordered sequential rules is that multiple rules can characterize the same situation. Partially-ordered rule mining generates fewer candidates, and fewer rules, by definition. Furthermore, they are described as more general, with a higher prediction accuracy than fully-ordered sequential rules, and they have been used in real applications [7]. In the proposed use case, describing a situation does not necessarily require an order, but the multiplicity of an element can be important. To explain this choice, we can take the example of a sound detection lamp: when one claps twice, i.e. when one makes the same sound twice, the lamp lights up.

3 Related Work

As said before, the system must search for partially-ordered prediction rules over a time series of elements coming from sensors and actuators. Thus, in the state of the art, two major areas of research should be considered: rule mining on time series and partially-ordered rule mining. Let us first recall some definitions. A prediction rule $R : E_c \Rightarrow E_p$ must be frequent and reliable. In rule mining, to check that a rule is frequent, its **support** is calculated. The notion of support depends on the structure of the input, but estimates the frequency of a rule, a set of elements, or an element. To ensure that a rule is reliable, its **interest** is calculated. Several measures can estimate the interest of a rule. The most known is confidence [3], but alternatives exist, such as conviction [3], lift [3] or netconf [1]. These measures depend on the supports of R, E_c and E_p.

3.1 Rule Mining on Time Series

[5] proposes a system mining basic rules on a sequence of elements, where one element predicts another. Those elements represent simple variations of stock market data. It can also search for more complex rules, where the condition is a sequence. This system therefore makes it possible to mine prediction rules over a time series. However, it seeks fully-ordered prediction rules, rather than partially-ordered ones. Also, the prediction part of the rules is limited to a single element, a limitation that we want to avoid in our AmI system. [11] can be considered as an improvement over [5], because this system looks for rules where the prediction is not limited to a single element. But, since it seeks fully-ordered rules, this system cannot be applied in our case. [10] introduces a notion of support for a time series, via a sliding window with a determined duration. The support of an element, a set of elements or a rule is the number of windows in which this element, set or rule appears. This algorithm finds partially-ordered rules, first finding sets of elements that are frequent, then combining these sets to generate rules. Other algorithms use this notion of support, including [6] which finds rules whose prediction is composed of one single element. The algorithm presented in [10] can therefore be applied in our case. But this definition of the support can be problematic: the elements of E_p being strictly later than E_c, the number of windows covering the rule R will be strictly lower than the number covering E_c. Even if E_p always appears after E_c, the support of the rule will be lower than

that of E_c, reducing its interest. Also, since the search is structured in two steps (mine frequent sets, then search for rules), the algorithm is not fully efficient.

3.2 Partially-Ordered Rule Mining

To our knowledge, few algorithms of partially-ordered sequential rule mining exists. The most known are RuleGrowth [8], and its variations, TRuleGrowth (TRG) [8] and ERMiner [7]. These algorithms take as input a set of transactions. A transaction is a time-ordered sequence of itemsets, but, unlike time series, without associated time stamp. RuleGrowth searches directly for prediction rules, unlike [10] that searches for frequent itemsets and then searches for rules on these itemsets. In addition, the incremental architecture of this algorithm allows to limit the size of the searched rules, and to limit the elements in which rules are searched. In our use case, we want to mine rules whose predictions are made only of elements from actuators. RuleGrowth allows this limitation directly during the search, reducing the total computation time. TRG is an extension of RuleGrowth that accepts the constraint of a sliding window, determined by a number of consecutive itemsets. It allows to limit the search to rules that can only occur in this window. ERMiner is presented as a more efficient version of RuleGrowth, but without an extension that accepts a sliding window. But those algorithms have a major problem in the proposed use case: they take transactions instead of a time series. The notion of support depends directly on the structure of transactions, and cannot be applied on a time series. Despite the advantages of these algorithms, they cannot be applied directly to our input data.

3.3 Scientific Problems

To our knowledge, the state of the art algorithms are not satisfactory enough to solve the initial problem. Two major issues need to be solved:

1. How to define the support of a rule in a time series that avoids the problem encountered in Sect. 3.1?
2. How to build a rule mining algorithm upon this new support measure?

In addition, this algorithm must address the following:

3. How to limit the duration of the found rules?
4. How to limit the search to certain elements in the condition or prediction?
5. How to avoid that a rule is found twice?

RuleGrowth answers points 4 and 5, but only takes transactions as input. Its extension, TRG, uses a sliding window that can be used to answer to the third problem with some modifications. The following section describes an adaptation of the AmI data to be accepted by TRG, and raises limitations of this adaptation. After, we describe our algorithm: TSRuleGrowth. It uses the principles of TRG, but applies them to time series, to deal with the first two problems.

3.4 Adapting Time Series to TRuleGrowth

To solve the problem of the input data of these algorithms, one can simply convert the time series into a list of transactions, as in Fig. 2. To do this, this time series is divided (1 in the figure) into smaller ones with a defined duration noted Δ_{tr} (2 and 3 in the figure). Then the notion of time of the small time series is removed, to keep only the order of appearance of the elements (4 in the figure). Without this notion of time, they are no longer time series, but rather sequences of elements, in other words, transactions.

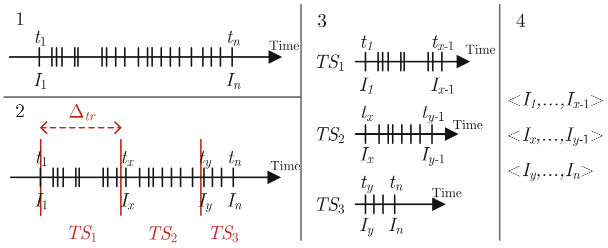

Fig. 2. Example of conversion of a time series into transactions

But the main problem of this implementation is the calculation of the support of a rule. Let us take the following example with three transactions:

$$\langle\{x\},\{x\},\{y\},\{x\},\{x\}\rangle$$
$$\langle\{x\},\{y\},\{x\},\{x\},\{x\}\rangle$$
$$\langle\{x\},\{x\},\{y\},\{x\},\{x\}\rangle$$

Here, $x \Rightarrow y$ is considered valid, because its support is 3, the same as x and y. As long as a rule has only been seen once in a transaction, it is considered valid throughout that transaction, even if it could have been invalidated, as in the example: x can be seen without y after, in all the transactions. Cutting a time series into transactions can lead to rules that are validated by mistake. There are other problems, inherent in Δ_{tr}. Having a small Δ_{tr} can increase the risk of a rule being "split in two", i.e. whose occurrence is separated between two transactions, which reduces interest. Having a large Δ_{tr}, over a time series, can reduce the absolute support of the rules the system is looking for.

Converting a time series into a set of transactions can be applied in the proposed use case. However, the above limitations have led us to create a new algorithm, inspired by TRG, which is fully adapted to time series.

4 TSRuleGrowth

4.1 Inputs, Outputs

This paper outlines the proposed rule mining algorithm on a time series of discrete elements : TSRuleGrowth, for "Time Series RuleGrowth", abbreviated to

TSRG. This algorithm is incremental, and can limit the search to certain elements in the condition and prediction. TSRG takes as inputs:

- $TS = \langle(t_1, I_1), ..., (t_n, I_n)\rangle, I_1, ..., I_n \subseteq E$: A time series of discrete elements
- min_{sup}: The minimum absolute support for a rule to be frequent
- min_{int}: The minimum interest for a rule to be reliable
- $window$: A time frame in which the rules must occur

TSRG produces partially-sequential prediction rules using multisets, detailed in Sect. 2.2. In the proposed use case, the prediction part of the rules is only composed of elements coming from actuators. Since TSRG takes a time series as input instead of a list of transactions, some notions need to be redefined: the support, the interest, and how to record the occurrences of a rule.

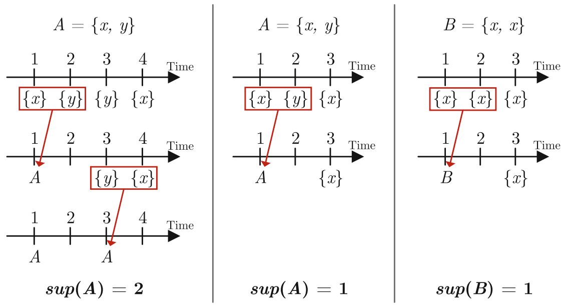

Fig. 3. Support calculation examples. Each column represents a step-by-step example of support calculation

Algorithm 1: Count : support counting algorithm

```
Data: $A$: multiset, $TS = \langle(t_1, I_1), ..., (t_n, I_n)\rangle$, $I_1, ..., I_n \subseteq E$: time series, $window$: duration
// Initialization
Assign a blacklist $b(a)$ to every unique element $a \in A$;
$sup(A) \leftarrow 0$;                                              // Support of $A$
// Sliding window through the time series
while the window has not reached the end of $TS$ do
    found ← True;
    Scan the window, record the time stamps of $a \in A$ in $T(a)$;
    foreach element $a \in A$ do
        $T(a) \leftarrow T(a) \setminus b(a)$;
        if $|T(a)| <$ multiplicity of $a$ in $A$ then
            found ← False ;                                   // No distinct occurrence
    if found is True then
        $sup(A)$ += 1;
        foreach element $a \in A$ do
            // Add the earliest time stamps of $T(a)$ to the blacklist of $a$
            $m \leftarrow$ multiplicity of $a$ in $A$;
            $b(a) \leftarrow b(a) \cup m$ earliest time stamps of $T(a)$;
    Slide the window by one itemset;
Return $sup(A)$;
```

4.2 Metrics

Support. For a time series TS noted $\langle(t_1, I_1), ..., (t_{n_s}, I_{n_s})\rangle$ where I_i is an itemset and t_i is an associated time stamp, the support of element x, noted $sup(x)$, is defined as the number of itemsets containing x (Eq. 1).

$$sup(x) = \big|(t_z, I_z) \in TS | x \in I_z\big| \tag{1}$$

The absolute support of a multiset of elements A is the number of distinct occurrences of all elements of A within the time window. If an occurrence of an element of A has contributed to an occurrence of the multiset A, it can no longer contribute to other occurrences of A. The examples in Fig. 3 can help to understand this concept more easily. The support counting algorithm, Count (Algorithm 1), scrolls a window on the time series. If all elements of A are seen, their occurrences will be blacklisted to prevent them from being involved in another occurrence of A. This ensures that the definition of the support is respected. If several occurrences of the same element of A are seen in the same window, only the earliest ones are blacklisted. It leaves newer ones the possibility to contribute to a future occurrence of A. The absolute support of $R : E_c \Rightarrow E_p$ is the distinct number of occurrences where all the elements of E_c are observed, followed by all the elements of E_p. The elements of E_c and E_p also have blacklists, grouped into two sets: one for the elements of E_c, and one for those of E_p. The relative support of an element x, a multiset A or a rule R, noted $relSup$, is its absolute support divided by the total number of itemsets in the time series (Eq. 2). This support can be applied to partially-ordered rules, unlike [5,11], and avoids the case expressed in Sect. 3.1.

$$relSup(R) = \frac{sup(R)}{|(t_z, I_z) \in TS|} \tag{2}$$

Interest. In TSRG, one can compute the interest of a rule through its confidence, conviction or lift as mentioned in Sect. 3.2. In the proposed use case, we chose netconf [1]. Unlike confidence, netconf tests the independence between occurrences of E_c and those of E_p. Also, unlike conviction and lift, it is bounded between -1 et 1, 1 showing that E_p has a high chance of appearing after E_c, -1 that E_p has a high chance of not appearing after E_c, and 0 that this chance is unknown. For a rule $R : E_c \Rightarrow E_p$:

$$netconf(R) = \frac{relSup(R) - relSup(E_c) \times relSup(E_p)}{relSup(E_c) \times (1 - relSup(E_c))} \tag{3}$$

4.3 Recording of Rule Occurrences

Let us take the example of $R : \{a, b, c\} \Rightarrow \{x, x, y\}$. An occurrence of R is decomposed as the occurrence of E_c and E_p. Indeed, an element can be found in both E_c and E_p, and it is necessary to distinguish the occurrences of this element in E_c from those in E_p. An occurrence of a multiset is recorded in an associative array, where the keys are the distinct elements of the multiset, and their values are the set of time stamps where the elements are observed. In Fig. 4, the occurrence of E_c is $\{a{:}\{2\}, b{:}\{2\}, c{:}\{1\}\}$ and the occurrence of E_p is $\{x\{5, 6\}, y{:}\{4\}\}$. Two time stamps are recorded for x, because it is present twice in E_p. To lighten the memory, an occurrence of a multiset can also be stored on a list of time stamps, provided that the multiset is ordered. On the list, the index of a time stamp is the same one as the index of the linked element in the multiset. In the previous example, the occurrence of $E_p = \{x, x, y\}$ is [5, 6, 4]. The recording of multiple occurrences of a multiset is a list of these structures. All occurrences of the rule are recorded in two lists, for E_c and E_p.

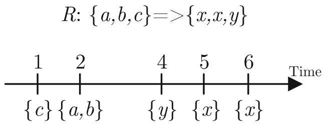

Fig. 4. Rule and time series example

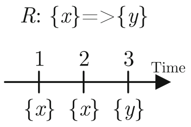

Fig. 5. Rule and time series example

4.4 Principles

Principles Shared with TRuleGrowth. TSRG takes the principles of TRG and applies them to time series. The algorithm uses a sliding window, to limit the search. But, unlike TRG where the window is a number of consecutive itemsets, TSRG has a time sliding window. It allows to restrict the search, and to have an estimate of the lifetime of a rule. Also, this algorithm will find basic rules, where one element can predict another. Then, recursively, it will extend them, by adding an element in E_c or E_p, via ExpandCondition and ExpandPrediction. This mechanism allows, if necessary, to limit the maximum length of the rules to be searched, i.e. the maximum number of elements in E_c and E_p. Then, TSRG applies two principles of TRG to avoid finding duplicate rules. First, ExpandPrediction cannot be called by ExpandCondition. Second, ExpandCondition and ExpandPrediction can add an element only if it is larger than all the elements of E_c or E_p, according to the lexicographic order.

New Principles. Let us take the example in Fig. 5. For this rule R, even if $sup(R) = 1$, two occurrences of the rule are possible: $\{x{:}\{1\}, y{:}\{3\}\}$ and $\{x{:}\{2\}, y{:}\{3\}\}$. This problem is inherent in time series: we cannot know *a priori* which occurrence will be useful for an extension of this rule. To do this, TSRG tries to extend all seen occurrences of this rule. In addition, TSRG does not use the same rule structure as TRG: instead of being sets, E_c and E_p are multisets. Therefore, a principle coming from TRG needs to be modified: ExpandCondition and ExpandPrediction can add an element if it is larger than all the elements of E_c or E_p, but also if it is equal to the greatest element of E_c or E_p, according to the lexicographic order. But a new problem of duplication arises. In Fig. 5, if we try to grow $\{x\}{\Rightarrow}\{y\}$ to $\{x, x\} \Rightarrow \{y\}$, the same occurrence will be found twice. $\{x{:}\{1\}, y{:}\{3\}\}$ will extend to $\{x{:}\{1, \mathbf{2}\}, y{:}\{3\}\}$, by adding the time stamp 2, and $\{x{:}\{2\}, y{:}\{3\}\}$ will extend to $\{x{:}\{\mathbf{1}, 2\}, y{:}\{3\}\}$, by adding the time stamp 1. To avoid this, TSRG does the following: if the rule extends to the greatest element of E_c or E_p, it should only record the time stamps of that element that occur **strictly later** than the last time stamp of that element in the base rule. Thus, in the previous example, the first occurrence is recorded, not the second.

4.5 Algorithm

Main Loop. Like TRG, the main loop (Algorithm 2) tries to find basic rules, i.e. rules whose conditions and predictions are composed of only one element. To do this, it computes the support for all basic rules that can be created in the time series. If one of these rules has a support higher than min_{sup}, it tries to make it grow, by adding an element in E_c (ExpandCondition), and in E_p (ExpandPrediction). Finally, it computes its interest for validation. As mentioned earlier, the algorithm computes all distinct occurrences of the rule for its support, but also all possible occurrences for the expansion of the rule. To do this, TSRG uses a blacklist system to discern occurrences. Multiprocessing can be added to TSRG, by treating all basic rules in parallel, to reduce the execution time.

Algorithm 2: TSRuleGrowth

```
Data: $TS$: time series, $min_{sup}$: minimum support, $min_{int}$: minimum interest, $window$:
      duration
1  Scan $TS$ once. For each element $e$ found, record the time stamps of the itemsets that
   contains $e$ in $T(e)$;
   // Creation of basic rules
2  foreach pair of elements $i$, $j$ do
3      $sup(i \Rightarrow j) \leftarrow 0$;                                  // Support of the rule
4      $O_c(i \Rightarrow j), O_p(i \Rightarrow j) \leftarrow []$;        // Occurrences of the condition and the prediction
5      $b(i), b(j) \leftarrow \emptyset$;                                              // Blacklists
6      foreach $t_i$ in $T(i)$ do
7          foreach $t_j$ in $T(j)$ do
8              if $0 < t_j - t_i \leq window$ then
                   // New occurrence of the rule
9                  Add $t_i$ to $O_c(i \Rightarrow j)$;
10                 Add $t_j$ to $O_p(i \Rightarrow j)$;
11                 if $t_i \notin b(i)$ and $t_j \notin b(j)$ then
                       // New distinct occurrence
12                     $sup(i \Rightarrow j)$ += 1;
13                     $b(i) \leftarrow b(i) \cup \{t_i\}$;
14                     $b(j) \leftarrow b(j) \cup \{t_j\}$;
       // Growth of basic rules
15     if $sup(i \Rightarrow j) \geq min_{sup}$ then
16         Run ExpandCondition and ExpandPrediction on the rule $i \Rightarrow j$;
17         if $netconf(\frac{|T(i)|}{|TS|}, \frac{|T(j)|}{|TS|}, \frac{sup(i \Rightarrow j)}{|TS|}) \geq min_{sup}$ then output rule
```

Algorithm 3: ExpandPrediction

```
Data: $TS$: time series, $E_c \Rightarrow E_p$: rule, $sup(E_c)$, occurrences of $E_c \Rightarrow E_p$, $min_{sup}$: minimum
      support, $min_{int}$: minimum interest, $window$: duration
// Growth of the original rule $E_c \Rightarrow E_p$
for each occurrence of the rule $E_c \Rightarrow E_p$ do
    foreach element $k$ seen in the search area do
        if $k$ has never been seen before then
            Create a new rule $E_c \Rightarrow E_{pk}$, its lists of occurrences and its blacklists;
            $sup(E_c \Rightarrow E_{pk}) \leftarrow 0$;
        foreach time stamp of $k$ $t_k$ inside the window (ascending order) do
            if $k > \max(e), e \in E_p$ or $t_k >$ occurrences of $k$ in the prediction part of the
            rule then
                Create a new occurrence of $E_c \Rightarrow E_{pk}$;
                if time stamps are not in the blacklists then
                    $sup(E_c \Rightarrow E_{pk})$ += 1;
                    Add the time stamps to the blacklists;
// Growth of the new rules found
foreach item $k$ where $sup(E_c \Rightarrow E_{pk}) \geq min_{sup}$ do
    $sup(E_{pk}) \leftarrow Count(E_{pk}, TS, window)$;
    Run ExpandCondition and ExpandPrediction;
    if $netconf(\frac{sup(E_c)}{|TS|}, \frac{sup(E_{pk})}{|TS|}, \frac{sup(E_c \Rightarrow E_{pk})}{|TS|}) \geq min_{int}$ then output rule
```

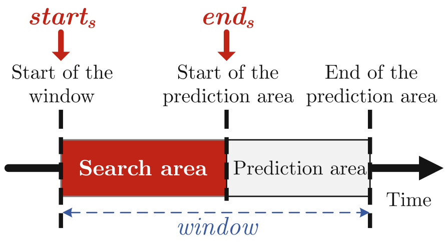

Fig. 6. ExpandCondition search area

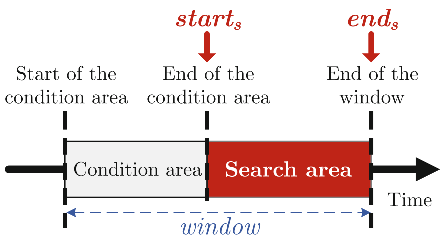

Fig. 7. ExpandPrediction search area

Expanding the Rules. ExpandCondition (Algorithm 3) tries to expand a rule by adding an element to its condition. It goes through all the possible occurrences of the rule, from the earliest to the most recent. To respect the time constraint imposed by *window*, the condition of a rule can only expand between two time stamps, noted $start_s$ and end_s, as seen in the Fig. 6. As for ExpandCondition, ExpandPrediction searches for new elements for E_p in the area described in Fig. 7. After having found new rules, ExpandCondition and ExpandPrediction try to grow them again, and verify their interest. Here, the simplified pseudocodes of TSRG and ExpandPrediction are described.

5 Experiments and Results

5.1 Results of TSRuleGrowth on Two Databases

We tested this algorithm on two databases: ContextAct@A4H (A4H) [9] and Orange4Home (O4H) [4]. Both databases contain daily activities of a single occupant. The characteristics of these databases and the parameters applied to TSRG

Table 1. Database characteristics, and parameters applied to TSRuleGrowth

	ContextAct@A4H	Orange4Home
Recording period	7 days in July and 21 days in November	4 consecutive weeks
Number of connected objects	213	222
Data records	35634	746745
TSRuleGrowth parameters		
min_{sup}	7	20
min_{int}	0.9	0.9
window (in seconds)	1,2,5,20,40,60,80,100,120, 140,160,180	1,2,5,10,15,20,25,30

are described in Table 1. The A4H database is located on the same physical location as O4H, but it has differences: the objects, as well as their names, are not the same. Also, the observed person is different, as is the observation period. Thus, the observed habits are different from one database to another. Some objects were specified manually to be actuators: shutters, doors, and lights for example. In addition, an amplitude discretization process was carried out on objects that reported continuous data, such as a temperature sensor. As a reminder, only actuators can provide elements for the predictive part of the rules. Also, the timestamps have been rounded to the nearest second on both databases. TSRG has been implemented in Python with multiprocessing[1]. First, let us look at TSRGs results on the two databases. Two aspects of the algorithm are evident in Fig. 8. The execution time and the number of rules increase exponentially with the window size. Indeed, when the window is larger, so is the search space. Thus, TSRG considers more and more elements, exponentially increasing the number of possible rules. For example, on the O4H database, 43 rules are observed on a one-second window, and 57103 on a 30-s window. This is explained by two complementary reasons. In a connected environment, several objects can be used to characterize a situation. For example, a person's entry into their home can be observed by a presence, noise, or door opening sensor. Thus, the rules can be formed from a combination of elements of these three objects. The larger a window is, the more combinations are possible, thus increasing the number of rules. Also, the rules discovered on a given window will, for the most part, be found again on larger windows, which also contributes to the increase in the number of rules. It is mentioned "for the most part" in the previous sentence, as some rules can be invalidated from one window to another. The invalidation of these rules does not come from their support, which can only increase from one window to another, but rather from their interest.

[1] CPU: Intel(R) Xeon(R) Gold 5118 @ 2.30 GHz, RAM: 128 GiB, Ubuntu 18.04.2 LTS.

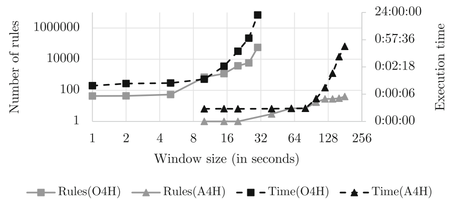

Fig. 8. Number of rules and execution time, TSRuleGrowth on O4H and A4H

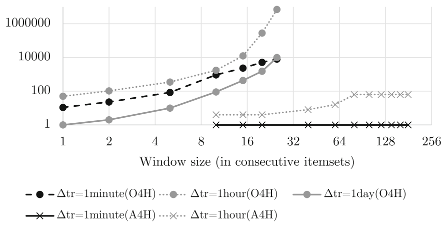

Fig. 9. Number of rules for TRuleGrowth on O4H and A4H

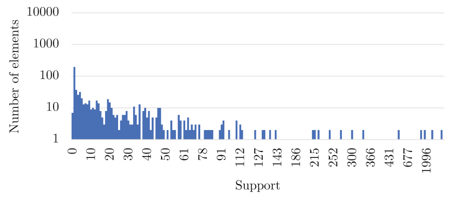

Fig. 10. Histogram of the elements grouped by their support in O4H

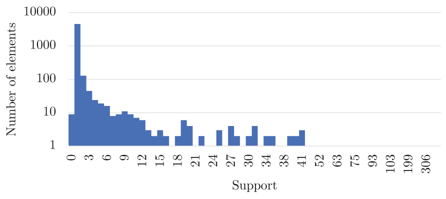

Fig. 11. Histogram of the elements grouped by their support in A4H

Indeed, the interest of a rule $R : E_c \Rightarrow E_p$ is calculated according to the support of R, E_c and E_p. In some cases, the support of E_c or E_p may increase more than R, reducing the interest enough to invalidate R. For example, the rule {'bathroom light 1: on, bathroom switch top left: off'} $\Rightarrow$ {'bathroom door: closed'}, discovered on O4H, has been validated on a window of 5 s. The supports are 38 for the rule, 42 for E_c and 91 for E_p. Its interest is therefore 0,904. By passing over a window of 10 s, the supports are still at 38 for the rule and 91 for E_p, but change to 44 for E_c. This is typically the case explained above, where the support of the rule increases less than that of its components. As a result, its interest drops to 0.863, invalidating the rule for this window. These invalidated rules represent only a fraction of the total number of discovered rules. Indeed, on O4H, by passing from a window of 10 to 15, 3 rules were invalidated, while 694 rules were observed on the window of 10 s, and 1170 on 15 s. In Fig. 8, we have explained the overall results curve of TSRG. However, this figure shows a very clear difference in results between O4H and A4H, although the physical environment and most of the connected objects are the same between these two databases. Two factors explain this. First, there is much less input data in A4H than O4H, as seen in Table 1: 35634 vs 746745. The less input data there is, the lower the probability of finding rules. Second, of all the elements in the database, very few are frequent in A4H. It can be observed by comparing Fig. 10 and 11. On A4H, there are 63 elements with an absolute support larger or equal to 20, unlike 395 on O4H. That is why we lowered min_{sup} to 7 on A4H, to have enough frequent elements (here, 132). This is reflected in the results reported by TSRG: even with min_{sup} lowered to 7, far fewer rules are found on A4H than on O4H

(1 vs 3689 for a 20 s window), and the execution is also faster (1 s vs 14 min for a 20 s window).

Let us now look at the rules themselves, first from the O4H database. On small windows (less than 5 s), straightforward rules are discovered, mostly the actions of switches in the environment. For example, {'bedroom switch bottom left: on'} ⇒ {'bedroom shutter 1: closed', 'bedroom shutter 2: closed'} and {'bedroom switch top right: on'} ⇒ {'bedroom light 1: off', 'bedroom light 2: off'}, seen in a 1 s window, indicate the different functions of the connected switches of the bedroom. Then, by increasing the window size, more complex rules are observed, characterizing the user's usual situations. {'office door: open', 'office presence: on', 'office switch left: on'} ⇒ {'office door: closed'}, seen in a 30 s window, indicates the user's entry into his office, by considering several different objects. For the A4H database, many fewer rules are observed in general, but some interesting rules are emerging. For example, {'fridge door: open'} ⇒ {'fridge door: closed'} describes that the fridge door will be closed within 40 s of being opened. As users, this rule may seem trivial to us. However, it should be remembered that the system has no preconceptions about the objects to which it is connected. With TSRG, the system learns the rules that govern the environment, and the habits of users. Thus, for O4H as for A4H, TSRG reports interesting results. Let us now compare these results with those of TRG.

5.2 Comparison Between TRuleGrowth and TSRuleGrowth

In this section, we compare TSRG with TRG. To do this, we use the same input databases, O4H and A4H. These data have been converted into transactions through the process detailed in Sect. 3.4. Three sets of transactions were made, with $\Delta_{tr} = 1$ min, 1 h and 1 day. The same parameters were applied between TRG and TSRG for min_{sup}, min_{int}, and *window* sizes. TRG uses netconf as a measure of interest, and min_{sup} is absolute instead of relative, but no other changes are made to this algorithm: the window used is still a consecutive number of itemsets, instead of a duration for TSRG. This difference implies that for TRG, it is possible to find rules whose duration can go up to Δ_{tr}. This explains why TRG can find more rules than TSRG in some cases. For example, on O4H, for a window of 25 itemsets/seconds, and with $\Delta_{tr} = 1$ h, TRG finds 267007 rules, and TSRG only 5677. Figure 9 shows that, like TSRG, TRG finds more rules exponentially as the window expands. However, this figure also shows the impact that the size of Δ_{tr} has on the number of found rules. On O4H, and for a window of 25 consecutive itemsets, TRG finds 8028 rules if $\Delta_{tr} = 1$ min, 7052216 rules if $\Delta_{tr} = 1$ h, and 9851 rules if $\Delta_{tr} = 1$ day. These results can be interpreted as follows: when $\Delta_{tr} = 1$ min, the number of rules is limited by the short duration of the transactions. When $\Delta_{tr} = 1$ day, fewer transactions are made. This reduces the absolute support of the rules and thus limits the search of the latter. Figure 9 shows that the number of rules made with $\Delta_{tr} = 1$ day catches up with that of $\Delta_{tr} = 1$ min as the window grows, until it exceeds it when window = 25. Many identical rules are observed by both TRG and TSRG. For a 1 s window/itemset, and a 1-h Δ_{tr}, 42 rules are common to these two algorithms. This represents

84% of the rules found by TRG and 98% of TSRG's rules. For the same $\varDelta_{tr}$, and a window of 10 itemsets/15 s, 1000 rules are common, i.e. 73% of the TRG rules, and 85% of the TSRG rules. But $\varDelta_{tr}$ can also limit the number of rules common to TRG and TSRG. Of all the possible window combinations, only 194 common rules are found at most for $\varDelta_{tr} = 1$ day, 2061 for $\varDelta_{tr} = 1$ min, and 8806 for $\varDelta_{tr} = 1$ h.

Why does $\varDelta_{tr}$ influence these results so much? The principles of Sect. 3.4 can explain this. The rules found with $\varDelta_{tr} = 1$ min are limited by the size of the transactions, while those with $\varDelta_{tr} = 1$ day are limited by their absolute support. Also, some rules can be validated by mistake. For example, {'staircase switch left: on'} ⇒ {'walkway light: off'} is seen with $\varDelta_{tr}$ of 1 h and 1 day, but not on $\varDelta_{tr} = 1$ min nor TSRG. Instead, the rule {'walkway switch 2 top right: on'} ⇒ {'walkway light: off'}, discovered by TSRG, is more coherent, because the two involved objects are in the same room. $\varDelta_{tr}$'s limitations are more visible on A4H. With $min_{sup} = 7$, and a $\varDelta_{tr}$ of 1 h or 1 day, TRG does not find any rule, for any window. If $\varDelta_{tr} = 1$ min, TRG finds a single rule, which is also observed by TSRG. Thus, in the case of A4H, TRG finds much fewer rules than TSRG for the same min_{sup}. By lowering min_{sup} to 6, TRG finds more rules: if $\varDelta_{tr} = 1$ day, no rule is found, if $\varDelta_{tr} = 1$ min, only 1 rule, also found by TSRG. This number can be up to 64 if $\varDelta_{tr} = 1$ h. It is higher than TSRG can find (maximum 40), but few rules are common to both TRG and TSRG (maximum 16). This is explained by the difference in the window concept between TRG and TSRG, giving unique rules to TRG, and the decrease in absolute support caused by $\varDelta_{tr}$, giving unique rules to TSRG.

We can therefore confirm that converting a time series into transactions can severely limit the search for rules and can create rules validated by mistake. TSRG, considering directly a time series, overcomes those shortcomings.

6 Conclusion

This paper described two contributions: a new notion of absolute and relative support over a time series, and an algorithm for searching partially-ordered prediction rules on a time series of discrete elements. The notion of support is freed from the limitations expressed in the state of the art, and the algorithm also distinguishes itself by its features: first, an incremental architecture, inspired by TRuleGrowth, allowing to limit the search to certain elements if necessary, as in the proposed use case; secondly, a sliding window, allowing to limit the duration of the searched rules; finally, the use of multisets in the rule structure, instead of sets in TRuleGrowth. TSRuleGrowth was tested on real data, from two databases of connected environments. The observed rules characterize short-term predictions, such as the action of a switch, and mid-term predictions, characterizing habits. These prediction rules make it possible to offer relevant automation possibilities to users of an AmI system. A comparison with TRuleGrowth has been made, highlighting problems and limitations of transactional rule mining algorithms on time series, which are not encountered on TSRuleGrowth. TSRuleGrowth finds prediction rules through connected objects data, and can evolve to take into account spacial aspects of the objects for example. But for an AmI system to be truly customized, it must consider the needs of users. A future version of such a system could therefore take into account their tastes to choose and classify the found rules, in order to display them in a way that is useful to them. Combined together, the relevance and usefulness of the rules will form a solid foundation for an AmI system.

References

1. Ahn, K.I., Kim, J.Y.: Efficient mining of frequent itemsets and a measure of interest for association rule mining. J. Inf. Knowl. Manage. **03**(03), 245–257 (2004). https://doi.org/10.1142/S0219649204000869
2. Augusto, J.C., McCullagh, P.: Ambient intelligence: concepts and applications. Comput. Sci. Inf. Syst. **4**(1), 1–27 (2007)
3. Azevedo, P.J., Jorge, A.M.: Comparing rule measures for predictive association rules. In: Kok, J.N., Koronacki, J., Mantaras, R.L., Matwin, S., Mladenič, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 510–517. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74958-5_47
4. Cumin, J., Lefebvre, G., Ramparany, F., Crowley, J.L.: A dataset of routine daily activities in an instrumented home. In: 11th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAm I), November 2017
5. Das, G., Lin, K.I., Mannila, H., Renganathan, G., Smyth, P.: Rule discovery from time series. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, KDD 1998, pp. 16–22. AAAI Press (1998)
6. Deogun, J., Jiang, L.: Prediction mining – an approach to mining association rules for prediction. In: Ślęzak, D., Yao, J.T., Peters, J.F., Ziarko, W., Hu, X. (eds.) RSFDGrC 2005. LNCS (LNAI), vol. 3642, pp. 98–108. Springer, Heidelberg (2005). https://doi.org/10.1007/11548706_11
7. Fournier-Viger, P., Gueniche, T., Zida, S., Tseng, V.S.: ERMiner: sequential rule mining using equivalence classes. In: Blockeel, H., van Leeuwen, M., Vinciotti, V. (eds.) IDA 2014. LNCS, vol. 8819, pp. 108–119. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12571-8_10
8. Fournier-Viger, P., Wu, C.W., Tseng, V.S., Cao, L., Nkambou, R.: Mining partially-ordered sequential rules common to multiple sequences. IEEE Trans. Knowl. Data Eng. **27**(8), 2203–2216 (2015). https://doi.org/10.1109/TKDE.2015.2405509
9. Lago, P., Lang, F., Roncancio, C., Jiménez-Guarín, C., Mateescu, R., Bonnefond, N.: The ContextAct@A4H real-life dataset of daily-living activities. In: Brézillon, P., Turner, R., Penco, C. (eds.) CONTEXT 2017. LNCS (LNAI), vol. 10257, pp. 175–188. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57837-8_14
10. Mannila, H., Toivonen, H., Inkeri Verkamo, A.: Discovery of frequent episodes in event sequences. Data Min. Knowl. Discov. **1**(3), 259–289 (1997). https://doi.org/10.1023/A:1009748302351
11. Schlüter, T., Conrad, S.: About the analysis of time series with temporal association rule mining. In: 2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), pp. 325–332, April 2011. https://doi.org/10.1109/CIDM.2011.5949303