Fast Extraction of Locally Optimal Patterns Based on Consistent Pattern Function Variations

Frédéric Pennerath

Supélec
2 rue Edouard Belin
F-57070 Metz, France
frederic.pennerath@supelec.fr

Abstract. This article introduces the problem of searching locally optimal patterns within a set of patterns constrained by some anti-monotonic predicate: given some pattern scoring function, a locally optimal pattern has a maximal (or minimal) score locally among neighboring patterns. Some instances of this problem have produced patterns of interest in the framework of knowledge discovery since locally optimal patterns extracted from datasets are very few, informative and non-redundant compared to other pattern families derived from frequent patterns. This article then introduces the concept of variation consistency to characterize pattern functions and uses this notion to propose GALLOP, an algorithm that outperforms existing algorithms to extract locally optimal itemsets. Finally it shows how GALLOP can generically be applied to two classes of scoring functions useful in binary classification or clustering pattern mining problems.

1 Introduction

Pattern mining consists in searching in some dataset relevant patterns in relation to some knowledge extraction problem. Formally, a *family of patterns* may be modeled as any partially ordered set $(\mathcal{P}, \leq_{\mathcal{P}})$ and a *dataset* as a multiset $\mathcal{D} \subseteq \mathcal{P}$ of patterns representing objects or observations. Then a pattern P is said to *describe* or *cover* a datum $d \in \mathcal{D}$ if $P \leq_{\mathcal{P}} d$. When objects are described by a set $\mathcal{I}$ of *items*, patterns are subsets of $\mathcal{I}$ called *itemsets*, ordered by set inclusion $\leq_{\mathcal{P}} = \subseteq$. Whatever their type (itemsets, sequences, graphs,...), datasets are intended to be analyzed by experts of the application domain in the hope of revealing some new pieces of knowledge. It is thus essential that these patterns are simultaneously characteristics of data, application-relevant, non-redundant, and as few as possible to make the analysis of these patterns practicable.

With respect to these requirements, many pattern mining methods generate too many patterns sharing too much similarity so that in practice, a study of these patterns is an annoying or even impossible task. This is a well-known problem with *frequent patterns* [1], i.e patterns P such that the proportion of data covered by P in dataset $\mathcal{D}$, called *relative frequency* $\sigma(P)$, is not less than some threshold $\sigma_0 \in [0, 1]$. But this problem also applies to many other pattern

J.L. Balcázar et al. (Eds.): ECML PKDD 2010, Part III, LNAI 6323, pp. 34–49, 2010.

families like emerging patterns for classification [2], or even condensed representations of frequent patterns like frequent closed patterns [3] or frequent free patterns [4] that are not much fewer than frequent patterns in many applications. A postprocessing step is thus required to select a subset of these patterns. It generally consists in computing for each pattern a non-monotonic score combining frequency with other pattern characteristics like length, classification measures, etc. The list of patterns is then sorted in descending order of score and only the few first hundred patterns of this list (also called top-k patterns [5]) are actually analyzed by experts. However this approach does not address the problem of redundancy: as similar patterns are likely to have similar scores, the top ranked patterns are likely to be considered by experts as clones of the same pattern (see examples of section 2.1 in [6]).

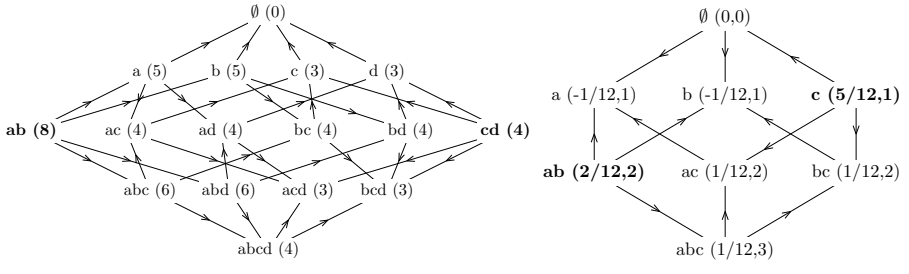
The family of *Most Informative Patterns*, or MIPs, has been introduced in [6] to address previous requirements similarly to pattern teams [7] and other global models like [8,9]. MIPs are defined as patterns maximizing locally some scoring function, that is, patterns whose score is not smaller than scores of neighboring patterns. Neighbors of a pattern P are defined as the set of *immediate predecessors and successors* of P, i.e. the set of every pattern P' such that there exists no other pattern included in between P and P'. MIPs are interesting for two reasons: first some scoring function estimates the value of a pattern specific to the application, second the local maximum criterion removes pattern redundancy and drastically reduces the number of selected patterns. The way this criterion removes pattern redundancy corresponds to the way experts are likely to select non-redundant patterns within the list of patterns sorted by descending order of score (see section 2.1 in [6]). The associated data-mining problem then consists in extracting every frequent MIP from a dataset. However the brute-force algorithm to extract frequent MIPs requires much more processing time than the extraction of frequent patterns as i) contrary to pattern frequency, scoring functions are not assumed to have specific properties enabling efficient pruning strategies, ii) while mining algorithms generate every frequent pattern only once, MIP extraction potentially requires to generate every pair of frequent neighboring patterns. A faster and more scalable extraction algorithm has been proposed in [6] but its performance is still two orders of magnitude slower than best frequent pattern mining algorithms like FP-growth [10]. This gap of performance prevents a complete extraction of MIPs for low frequency thresholds.

While the MIP model [6] considers some specific class of scoring functions whose properties ensure MIPs are "clustering patterns", i.e. descriptive of substantial fractions of datasets, the key idea of using the local maximum criterion to remove redundancy may serve other purposes. For this reason, the present article considers the more general problem of extracting *Locally Optimal Patterns* relatively to any pattern scoring function. The main contribution of this paper is the introduction of an algorithm called GALLOP (for Generic ALgorithm to extract Locally Optimal Patterns) that outperforms with typically one order of magnitude prior algorithm of [6] to extract MIPs when patterns are itemsets. Moreover the article shows GALLOP's performance generalizes to another type

of scoring function suiting binary classification problems. More fundamentally, this article contributes to provide useful concepts to characterize pattern scoring functions in the same way real analysis in mathematics provides useful concepts to characterize real functions. In particular this article introduces concepts of *consistent variation* and *dominance influence*, on which GALLOP's heuristics are based. The rest of this paper is structured as follows: section 2 formally defines Locally Optimal Patterns, section 3 introduces GALLOP and its heuristics based on both aforementioned concepts, section 4 analyzes results of tests run on reference datasets and section 5 concludes.

2 Locally Optimal Patterns

For the sake of generality, no restriction is made at this stage on the type of considered patterns (itemsets, sequences, graphs...). Therefore let a *pattern space* $\mathcal{P}$ be a set of patterns ordered by some partial ordering relation $\leq_{\mathcal{P}}$. A *pattern scoring function* is any function $s : (\mathcal{P}, \leq_{\mathcal{P}}) \to (\mathbb{S}, \leq_{\mathbb{S}})$ mapping a pattern P to a *score* $s(P)$, where scores are elements of any set $\mathbb{S}$ ordered by some partial or total ordering relation $\leq_{\mathbb{S}}$. A simple example of scoring function is the *area function* s_a that maps an itemset P to a real number called *area*, that is, the product $s_a(P) = |P| \cdot \sigma(P) \cdot |\mathcal{D}|$ of pattern length $|P|$ and absolute frequency $\sigma(P) \cdot |\mathcal{D}|$ of P in some dataset. This function is useful in clustering problems as it privileges descriptive patterns representative of large fractions of datasets. This scoring function is illustrated on Fig. 1(a) by the lattice of itemsets made of items a, b, c, and d, ordered by inclusion and scored with their area. The



(a) Area s_a (b) Difference s_d

Fig. 1. Itemset lattices with score and dominance relation (an arrow $P_1 \to P_2$ means pattern P_1 dominates pattern P_2) for the area function (a) and the extended difference function (b). Locally optimal patterns appear in bold.

area of pattern P is computed from frequency of P in a dataset made of seven objects, whose descriptions are respectively a, b, ab, cd, abc, abd, and $abcd$. For instance, score of pattern ac is $s_a(ac) = |\{a, c\}| \times \sigma(ac) \times 7 = 2 \times 2 = 4$ since ac is a subset of data abc and $abcd$, and thus $\sigma(ac) = 2/7$.

Another example of scoring function is the *difference function* s_d privileging class discriminating patterns extracted from binary classification oriented data. This function maps pattern P to a real number $s_d(P) = \sigma_+(P) - \sigma_-(P)$ where $\sigma_+(P)$ and $\sigma_-(P)$ respectively denote relative frequencies of P in datasets of positive and negative examples of some target concept. Length of pattern P might optionally be appended to $s_d(P)$, so that scores are vectors $(s_d(P), |P|)$ ordered by lexicographic order (i.e. second dimensions of two scores are compared only if first dimensions are found equal). This second dimension is a refinement to privilege $\leq_{\mathcal{P}}$-maximal patterns within equivalence classes of patterns covering the same sets of positive and negative examples. Lattice on Fig. 1(b) illustrates this *extended difference function* assuming positive and negative examples are respectively data with and without item d in the aforementioned dataset. For instance, score of pattern ac is $s_d(ac) = (\sigma_+(ac) - \sigma_-(ac), |\{a, c\}|) = (1/3 - 1/4, 2)$. The length dimension allows to distinguish abc of length 3 from ac and bc of length 2 while all three patterns cover the same positive (i.e. $abcd$) and negative (i.e. abc) data and thus have the same frequency difference $1/3 - 1/4$.

The general objective that is pursued by the author is to get better insights about how any given pattern function s "behaves" in the pattern space on which s is defined. This article provides first concepts and methods to address this question by searching for patterns maximizing locally function s within their neighboring patterns. Two patterns are hereafter *neighbors* if one is the *immediate successor* of the other. A pattern P' is an *immediate successor* of a pattern P (denoted $P \prec_{\mathcal{P}} P'$) if $P <_{\mathcal{P}} P'$ and no pattern P'' exists such that $P <_{\mathcal{P}} P'' <_{\mathcal{P}} P'$. Pattern P is then an *immediate predecessor* of P'. Two itemsets P_1 and P_2 are thus neighbors if they differ with one item only, or equivalently, if their edit distance hereafter defined as the cardinality of their symmetric difference $d(P_1, P_2) = |P_1 \setminus P_2| + |P_2 \setminus P_1|$ is equal to one.

Definition 1. *Given a scoring function* $s : \mathcal{P} \to (\mathbb{S}, \leq_{\mathbb{S}})$*, a pattern* $p_1 \in \mathcal{P}$ *is said to* dominate *pattern* $p_2 \in \mathcal{P}$ *if and only if* p_1 *and* p_2 *are neighbors and* $s(p_1) >_{\mathbb{S}} s(p_2)$*. A* locally optimal pattern *or* LOP *is a pattern that is not dominated by any pattern. Then given an anti-monotonic predicate* p *defined over* $\mathcal{P}$*, a pattern is said* valid *if it satisfies* p*. The problem of the* extraction of valid locally optimal patterns *consists in extracting every optimal pattern relatively to* s *that is valid relatively to* p*.*

Figure 1 represents dominance by orienting lattice edges: an arc from pattern P_1 to pattern P_2 means P_1 dominates P_2. Locally optimal patterns, called *optimal* patterns for short, are patterns that are not pointed by any arc like patterns ab and cd on Fig. 1(a). The anti-monotonic predicate p is introduced to control the number of patterns to process as pattern spaces are generally very large or even infinite sets. In pattern mining applications, valid patterns are likely to be frequent patterns. Because p is a secondary parameter of the problem, it is important that the range of the dominance relation be not limited only to valid patterns but to every neighbor of every valid pattern (including these that are not valid). Otherwise the locally optimal character of valid patterns would depend on the arbitrary choice of predicate p, which is a negative side effect. For

instance on Fig. 1(a), pattern c of frequency 3 is not optimal as it is dominated by pattern ac of frequency 2. However if one arbitrarily sets threshold $\sigma_0 = 3$, pattern c appears not dominated by any other frequent pattern and could be mistakenly believed optimal.

3 Locally Optimal Pattern Extraction

LOP extraction algorithms have to proceed unit tasks called *pattern comparisons*, consisting in comparing scores of two neighboring patterns and discarding the dominated neighbor (if any) from the output set. LOP extraction is a problem of higher time and space complexities than a simple enumeration of patterns as used in frequent pattern mining, since the number of pattern comparisons may reach the number of pairs of neighboring patterns (i.e the number of edges in the diagram of order $(\mathcal{P}, \leq_{\mathcal{P}})$) instead of the number of patterns (i.e. the number of vertices in the order diagram).

3.1 Existing Algorithms

The brute-force extraction of LOPs described in [6] to extract frequent MIPs, consists in an exhaustive depth first order exploration of the pattern space starting from the empty pattern. This approach called *direct extraction* in [6] is clearly not scalable as it requires to memorize every investigated pattern (with its score and a selection flag) and it is also very slow for several reasons: in particular, the method systematically generates every invalid (e.g non-frequent) pattern having at least one valid predecessor. This set, called hereafter the *disjunctive negative border*, is in many applications much larger than the set of valid patterns itself and its generation requires a lot of processing time [6].

A better solution is proposed in [6] based on a level-wise filtering algorithm: in a first step, the method splits valid (e.g frequent) patterns into levels $(L_n)_{n\geq 0}$ of patterns of size n, so that a pattern P in level L_n is provided to the second step, only if P is not dominated by any valid neighbor found in levels L_{n-1} or L_{n+1}. This way the second step, called *postfiltering*, only has to consider a small number of patterns called *candidates* to compare with their successors in the disjunctive negative border. The level-wise method is considerably faster than the direct extraction as it does not generate any pattern of the disjunctive negative border during the first step and only a small fraction of it during the second step. The method is also more scalable as it only stores one level of frequent patterns in memory at a time.

However, the method systematically compares every pair of valid neighboring patterns. Moreover it requires to split input patterns produced by fastest frequent pattern mining algorithms like FP-Growth into as many files as levels, implying many slow parallel file system accesses. In order to further reduce the gap between times to extract the list of frequent patterns and to extract optimal patterns from them, GALLOP has been designed with two goals: first, GALLOP avoids costly IO access times due to the reordering of many input patterns in

levels by processing sequentially the input pattern file in only one pass, while at the same time reducing memory needs. Second GALLOP reduces the number of useless pattern comparisons that do not discard any pattern from the candidate set. The next two subsections detail how GALLOP addresses each of these goals.

3.2 GALLOP's Pattern Sequential Processing

GALLOP retains the previous idea of a two-step processing to avoid the complete generation of the overlarge disjunctive negative border. Indeed for the full extraction process to be generic, it has been broken down in four successive steps:

1. The **mining step** first enumerates every valid pattern relatively to some predicate p, like for instance computing every frequent pattern with its frequency.
2. The **scoring step** sequentially processes valid patterns to score each of them relatively to some function s.
3. In the **filtering step**, GALLOP extracts candidates from scored patterns, later called *input patterns*.
4. The **postfiltering step** compares scores of candidates with those of their successors in order to determine which candidates are optimal.

As a consequence, GALLOP is a generic algorithm that only compares pattern scores and thus does not depend on s and p. Nevertheless GALLOP depends on the scoring order $(\$, \leq_\$)$. In practice the current implementation assumes scores are real vectors ordered by either lexicographic or product ordering relation. As already stated, one objective of GALLOP is to process the very large input pattern file in one single pass while remaining complete. Since the fastest frequent pattern mining algorithms like FP-growth used in the first step mentioned above enumerate itemsets in lexicographic order (assuming some implementation-dependent ordering of items), GALLOP processes input patterns in the same enumeration order thanks to a recursive procedure.

Definition 2. *A* pattern enumeration *is formally defined as a linear ordering relation* $\trianglelefteq$ *called* precedence relation *defined over the set of patterns, so that* $P_1 \triangleleft P_2$ *means "*P_1 *is enumerated before* P_2*". *P_1 *then* precedes P_2 *or conversely* P_2 succeeds P_1*. A lexicographic enumeration of itemsets has a precedence relation equal to some lexicographic ordering of itemsets :* $(i_1, \ldots, i_n) \triangleleft (i'_1, \ldots, i'_{n'})$ *if there exists a subscript* $j \geq 0$ *such that for all* $k \leq j$, $i_k = i'_k$ *and either* $j = n$ *and* $n < n'$ *or* $i_{j+1} <_\mathcal{I} i'_{j+1}$ *where* $<_\mathcal{I}$ *is an arbitrary linear ordering of items.*

Such enumeration may be efficiently implemented by extending recursively every pattern $P = (i_1, \ldots, i_n)$ to pattern $P' = (i_1, \ldots, i_n, i_{n+1})$ by appending the smallest item $i_{n+1} >_\mathcal{I} i_n$ not already appended to P. Pattern P' is one of the *children* of P and P is the *parent* of P'. In such enumeration, two neighboring patterns are said *lineal neighbors* if one pattern is the parent of the other. Otherwise they are said *transverse neighbors*. Thus every pattern P has up to four classes of neighbors as illustrated on Fig. 2: one lineal predecessor (or parent)

and several transverse successors that all precede P, and several lineal successors and transverse predecessors that all succeed P. This distinction between

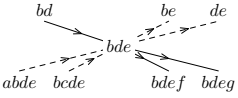


Fig. 2. Lineal and transverse neighbors of itemset *bde* wrt lexicographic enumeration of itemsets made of items from a to g ordered alphabetically: *bd* (parent), *bdef* and *bdeg* (children) are lineal neighbors. *be*, *de*, *abde* and *bcde* are transverse neighbors. An arrow from pattern P_1 to pattern P_2 means P_1 precedes P_2.

lineal and transverse neighbors is important as a sequential processing of patterns enumerated in some lexicographic ordering may compare current pattern P with its lineal neighbors at almost no computational cost since the score of P can be passed to recursive processing of its children. In contrast, comparison of P with a transverse predecessor P' raises a much more difficult problem as the number of patterns succeeding P and preceding P' may be arbitrarily large. However comparisons with transverse neighbors are essential to screen efficiently candidates as later shown in section 4. Figure 3 illustrates this importance: Only

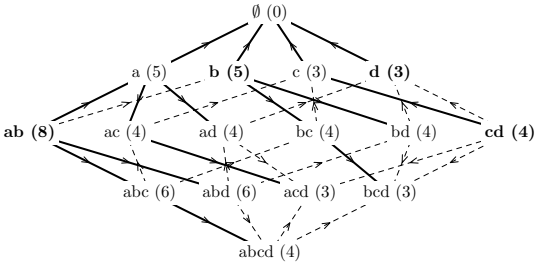


Fig. 3. Itemset lattice of Fig. 1(a) with lineal (thick) and transverse (dashed) neighboring. Bold patterns (i.e *ab*, *b*, *cd*, and *d*) are not dominated by any lineal neighbor.

comparisons with transverse neighbors are able to eliminate non-optimal candidates b and d among the four patterns that are not lineally dominated.

In order to compute efficiently comparisons with transverse neighbors, the sequential process of patterns enumerated in lexicographic order presents an interesting possibility: given some currently processed pattern P, the algorithm may "postpone" comparisons of P with its transverse predecessors until these predecessors (which succeed P) are in turn processed. This postponing can be implemented by inserting predecessors of P into a priority queue (i.e. binary heap) of patterns. The sorting order underlying the queue is the same lexicographic order used to enumerate input patterns, so that the top queue element is always the next pattern to process among all queued patterns. Every entry

in the queue carries the postponing pattern P, its score $s(P)$ and a flag to remember if P has already been found dominated. Then since P is valid and p is anti-monotonic, every transverse predecessor P' of P is valid and will eventually be processed. When P' becomes the currently processed pattern, GALLOP may learn whether some transverse successor of P' like P has previously postponed a comparison with P', by checking whether the top of the queue is P'. In this case, the top entry of the queue is popped, P and P' are compared before their flags are updated accordingly. In practice, since the processing of transverse predecessors of P occur in some deterministic order, only the next occurring of these predecessors is present in the queue. When this predecessor P' of P is processed, the next occurring transverse predecessor P'' of P is computed from P' and P's entry is pushed down to the position of P'' in the queue. This limits the size of the queue to the number of currently postponing patterns. Figure 4 summarizes the general principle of GALLOP's recursive procedure.

```
function filter(Current pattern P, Value V of P)
   (Integer nBT and priority queue Q are global variables)
   while(Q is not empty)
      Let (Pattern P',Value V') be the top of Q
      if(P' ≠ P) break;
      if(V.score <$ V'.score), V.lop ← false
      else if(V.score >$ V'.score), V'.lop ← false
      Be P'' the next predecessor of V'.pattern succeeding P.
      if(P'' is defined)
1        if((P'',V') is to be postponed), move (P'', V') in Q
      else
         if(V'.lop), output candidate (V'.pattern, V'.score)
         pop Q
      end if
   end while
   (Pattern P', Score S, nBT) ← readNextPattern()
   Be value V' with
      V'.pattern ← P', V'.score ← S, V'.lop ← true
   while(nBT = 0)
      if(V.score <$ V'.score), V.lop ← false
      else if(V.score >$ V'.score), V'.lop ← false
      call filter(P',V')
      nBT ← nBT - 1
   end while
2  if((P, V) is to be postponed),
      Be P'' the first transverse predecessor of P succeeding parent of P
      insert (P'',V) into Q
   end if
end
```

Fig. 4. Sketch of GALLOP's recursive procedure

Since postfiltering compares every candidate with its successors, GALLOP must only guarantee two conditions i) every candidate in the output is not dominated by any predecessor ii) every LOP is not discarded from the output. Therefore, a pattern known to be dominated might not be compared with every of its transverse predecessors. This freedom authorizes different strategies appearing in GALLOP's algorithm on lines 1 and 2, where GALLOP decides whether pattern comparisons must be postponed or not. Determining the best strategy is fundamentally a problem of balancing effort between GALLOP and

postfiltering: in other words, the fewer postponed comparisons, the faster GALLOP, but the many more candidates, and finally the slower postfiltering. This optimal strategy must lie between two extreme strategies:

Maximal effort strategy. Given current pattern P, an obvious strategy consists in postponing systematically comparisons of current pattern P even if P is already known to be dominated by some pattern preceding P. Like the level-wise algorithm, this maximal effort strategy guarantees that every pair of valid neighbors has been compared so that GALLOP's output is reduced to the minimal set of candidates, that is, valid patterns that are not dominated by any other valid pattern. The postfiltering is thus the fastest possible but systematic postponing is likely to require a lot of time and memory.

Minimal effort strategy. Conversely minimal effort strategy consists in stopping postponing comparisons with predecessors of current pattern P as soon as P appears dominated by some patterns. A pattern dominated by a lineal neighbor or a transverse successor is thus never inserted in the queue. This strategy provides the fastest possible version of GALLOP that still guarantees every candidate produced as output is not dominated by any predecessor. However this strategy is "selfish" as its only concern is pattern P: this strategy does not help to discard transverse predecessors dominated by P. Consequently this strategy is likely to produce many candidates and to require a long postfiltering.

3.3 GALLOP's Heuristics Based on Variation Consistency

The optimal strategy consists in making only comparisons which discard one of the two compared neighbors from the candidate set, that is, when one pattern is dominated by the other while it was not already known to be dominated by some preceding pattern. Such comparisons are called *useful*, all others are said *useless*. However the optimal strategy is not achievable as the decision that a pattern P has to be compared with some of its transverse predecessors must be made when P is currently processed, at a time no transverse predecessors are already known to be dominated. Therefore the processing of P may only guess, based on some heuristics, which comparisons with P's predecessors are useful and must be postponed in the queue. These heuristics must rely on some expected properties of the scoring function over the pattern space. While scoring functions are not required to have any theoretical property, pattern scores found in practical applications are expected to have some regular distribution, in the same way data points used in regression models are expected to be samples randomly scattered around some piecewise continuous function. The notion of *consistent variation* is hereafter introduced to model regularity existing in variations of pattern functions.

Definition 3. *A* diamond configuration *is a set of four patterns P_1, P_2, P_3 and P_4 such that $P_1 \prec_{\mathcal{P}} P_2 \prec_{\mathcal{P}} P_4$ and $P_1 \prec_{\mathcal{P}} P_3 \prec_{\mathcal{P}} P_4$ as illustrated on Fig. 5. Given such a diamond configuration and a scoring function s, variations of s*

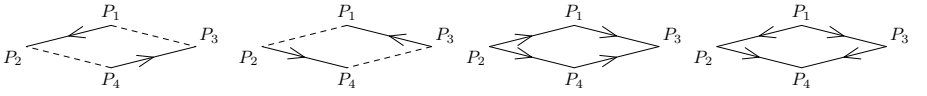


Fig. 5. Diamond configurations: examples of inconsistent (a and b) and consistent (c and d) variations. Arrows give dominance directions.

from P_1 to P_2 and from P_3 to P_4 are inconsistent *if P_1 dominates P_2 and P_4 dominates P_3 or if P_2 dominates P_1 and P_3 dominates P_4.*

Definition 4. *A scoring function has consistent variations within a set S of patterns if no diamond configuration within S has some inconsistent variation.*

Non-decreasing pattern functions (i.e. $\forall P_1, \forall P_2,\ P_1 \leq_{\mathcal{P}} P_2 \Rightarrow s(P_1) \leq_{\mathbb{R}} s(P_2)$) like pattern length, or conversely non-increasing functions like pattern frequency have by definition consistent variations over the whole pattern space. However consistent variation is still statistically true for more complex non-monotonous scoring functions like area or difference scoring functions as shown in Tab. 1. The

Table 1. Variation consistency ratio for different datasets

Dataset	Scoring function	Positive class	Negative class	Thres. σ_0	Number of diamonds	Consistency Ratio
Mushrooms	area			0.05	132 M	98 %
	diff.	edible	poisonous	0.05	59 M	97 %
Breast-Cancer	area			0.001	3.9 M	85 %
	diff.	cancerous	healthy	0.001	1.6 M	85 %
Vote	area			0.01	10 M	94 %
	diff.	republican	democrat	0.01	4.3 M	95 %
Chess	area			0.5	41 M	97 %
Connect	area			0.8	173 M	98 %

table provides the *consistency ratio* of both scoring functions s_a and s_d (when target classes are available) for some UCI datasets. This ratio is computed from a set of frequent patterns: for every possible diamond configuration of frequent patterns, consistency of both possible pairs of variations is tested. The ratio is defined as the number of consistent pairs over the total number of pairs. For every function and dataset (but Breast-Cancer), the ratio is at least 94 %. This observation leads to an alternative strategy called *lazy strategy*.

Comparison Pruning based on a Lazy Evaluation Strategy. This strategy consists in making comparisons of current pattern P with its transverse predecessors only if P is not dominated by any lineal neighbor. To understand why this strategy is sound, let some current pattern P be dominated by some lineal neighbor P_L. Figure 6 illustrates the case P_L is a predecessor of P. For every transverse predecessor P' of P, let i be the item such that $P = P' \cup \{i\}$ and let P'_L be the itemset such that $P_L = P'_L \cup \{i\}$. Then the four patterns builds

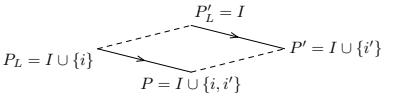


Fig. 6. Lazy strategy heuristic: if current pattern P is dominated by a lineal neighbor P_L (here a predecessor), every transverse predecessor P' is likely to be dominated by one of its lineal neighbor P'_L.

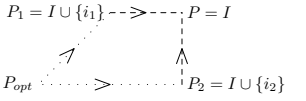


Fig. 7. *Consistent influence*: P is dominated by two transverse successors P_1 and P_2 because of a common influence of some remote locally optimal pattern P_{opt}. Dashed and dotted lines represent resp. transverse neighborhood and influence of P_{opt}.

a diamond where arcs $P \to P_L$ and $P' \to P'_L$ are parallel and have the same orientation in the order diagram. Assuming variations of the scoring function s are consistent and since P_L dominates P, P' is likely to be dominated by P'_L. As P is known to be dominated, comparing P with its transverse predecessors is thus likely to be useless. Implementing the lazy strategy when processing P is possible by delaying the decision to postpone comparisons with transverse predecessors after processing recursively lineal children of P (cf line 2 on Fig. 4). At that point, P is known to be lineally dominated or not.

More Comparison Pruning based on LOP Dominance Influence. The concept of variation consistency over current pattern P and its lineal neighbors leads to define the lazy strategy as an optimized version of the maximal strategy. Similarly the concept of *dominance influence* leads to define an optimized version of the lazy strategy.

Definition 5. *Given a locally optimal pattern P_{opt} relatively to some scoring function s and a set $\mathcal{E}$ of patterns, P_{opt} has a* dominance influence *over $\mathcal{E}$ if for every pair of neighbors $\{P, P'\}$ in $\mathcal{E}$ such that P is strictly closer to P_{opt} than P' (according to the edit distance d defined in Sect. 2), P dominates P'.*

The intuition behind this concept is that every local optimal pattern exerts a dominance influence over some surrounding pattern subspace: for example, optimal pattern *ab* on Fig. 1(a) exerts a dominance influence over all patterns but *cd*, *c*, and *d* while optimal pattern *cd* only exerts a dominance influence over *cd*, *c*, *d*, $\emptyset$ and *bcd*. This hypothesis is used to prune more useless comparisons in the case current pattern P is not lineally dominated but is under the dominance influence of some optimal pattern P_{opt} through a number of transverse successors $\{P_i, 1 \leq i \leq n\}$ such that $\forall i$, $P \prec P_i \subseteq P_{opt}$ and $P_1 \lhd \cdots \lhd P_n$ as illustrated on Fig. 7. Since P is not dominated lineally, it is likely that transverse successors P_i are also not dominated lineally, according to the hypothesis of variation consistency. Therefore the lazy strategy will postpone comparisons of every P_i.

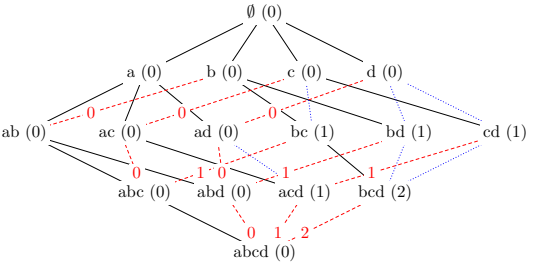


Fig. 8. Shifting technique for potential candidate $P_{cand} = abcd$. Red dashed, blue dotted, and black plain edges represent resp. postponed transverse, skipped transverse, and lineal comparisons. Number attached to itemset is its initial shift value, and number attached to edge representing a postponed comparison is the shift value of the edge successor once comparison occurs.

P will thus be compared n times with each P_i whereas one comparison with P_1 would have sufficed to discard P from the candidate set.

The proposed heuristic called *pattern shifting* aims at avoiding the $n-1$ useless comparisons with P by ensuring that for every *potential candidate* P_{cand} (i.e. a current pattern that has not been found dominated by any neighbor so far, and might potentially have a dominance influence over surrounding patterns), if $P \subset P_{cand}$, P will be compared with one and only one transverse successor P_k of P, such that $P \subset P_k \subseteq P_{cand}$. The heuristic expects a single comparison of P_k will suffice to discard P according to the dominance influence hypothesis. In practice k is chosen to be 1, so that pattern P_k is the first enumerated transverse successor of P such that $P_k \subseteq P_{cand}$ and is called *eldest successor*. For instance, if $P_{cand} = abcd$, then $P = ad$ has two transverse successors in the lattice of Fig. 3, that are abd and acd and its eldest successor is $P_1 = abd$. These eldest successors build a forest structure whose trees are built upward as shown by postponed comparisons (red dashed edges) on Fig. 8. Shifting is an efficient implementation of this forest structure starting from potential candidate P_{cand}. Given current pattern P, shifting determines which is the first transverse predecessor P', pattern P should be compared with (i.e should be the eldest successor of). All comparisons with transverse predecessors of P preceding P' may thus be ignored. The number of these skipped patterns is called *shift* and initialized to 0. In the optimal case, P receives a postponed comparison from one unique transverse predecessor which is the eldest successor P' of P. The shift of P is then initialized to the shift of P', and every time P is compared with a new transverse predecessor, its shift is incremented by one. However the shift of P is sometimes forced to be initialized to 0 when P appears to be a new potential candidate. This simple algorithm builds upward trees between transverse neighbors starting from potential candidates as illustrated on Fig. 8: assuming $abcd$ is a potential candidate, its initial shift is set to 0. This shift is incremented when comparing $abcd$ with abd, acd, and bcd. Initial shift for abd, acd, and bcd are

thus resp. 0, 1, and 2. When *acd* is processed as the current pattern, its shift is 1 and skips its first transverse predecessor *ad*, since *ad* is already compared with its eldest successor *abd*. When *bcd* is processed, its shift value is 2, and since it has only two transverse predecessors *bd* and *cd*, *bcd* does not postpone any comparison as it is not the eldest successor of any of its transverse predecessors. A consequence of discovering several new potential candidates is that a current pattern P can be compared with more than one eldest successor, with different interfering shift values. In this case, the most cautious choice is to skip a minimal number of postponed comparisons, by setting the shift of P to the minimal shift of eldest successors that postponed comparisons with P.

4 Tests and Empirical Analysis

Tests on some standard PC (Intel Core2 1.8GHz with 1.5GB RAM) have been performed on reference itemset datasets from the UCI repository in order to compare both processing time and scalability (measured by the maximum number of memorized patterns at a time) of each different algorithm: the prior level-wise algorithm presented in [6] and the four GALLOP's versions using the maximal, minimal, lazy strategy, and the lazy strategy with shifting (referred as GALLOP-Lazy+). All these algorithms have been used to extract frequent optimal patterns relatively to both area and difference functions. Table 2 summarizes test results. Whatever the dataset and scoring function are, the level-wise algorithm, GALLOP-Maximal, GALLOP-Lazy, and GALLOP-Lazy+ always appear in this order in the list of algorithms sorted in descending order of processing time. GALLOP-Lazy+ outperforms the level-wise processing time with at least one order of magnitude but for the *Connect* dataset. Speed improvements from GALLOP-Maximal to GALLOP-Lazy and from GALLOP-Lazy to GALLOP-Lazy+ attest the soundness of consistent variation and dominance influence heuristics. Since the level-wise algorithm and GALLOP-Maximal compare every pair of frequent neighbors, they always produce the same number of candidates. More surprising is that GALLOP-Lazy, while pruning many more comparisons than GALLOP-Maximal, produces the same number of candidates and thus does not slow down postfiltering: as a consequence, the lazy strategy noticeably reduces processing time and improves scalability of the filtering step without increasing time of the postfiltering step. This is also true about pattern shifting: even if GALLOP-Lazy+ sometimes produces few more candidates than GALLOP-Lazy, the time saved by pruning more comparisons largely balances the small extra time spent in postfiltering. These observations are confirmed for every threshold value as shown on Fig. 9(a) and (b). *Connect* is the only dataset where heuristics do not substantially reduce the total processing time. This is because postfiltering has a time complexity in $\Theta(|\mathcal{D}| \cdot |C|)$, proportional to the number of candidates $|C|$ and to the very large size $|\mathcal{D}|$ of dataset *Connect*. Since every GALLOP's version but GALLOP-minimal produces the same 214 candidates and since postfiltering of these candidates is about 500 longer

Table 2. Comparison of algorithm performances for different datasets, scoring functions, and frequency thresholds. Every test is described by total processing time (including time for filtering input patterns and postfiltering candidate patterns), ratio of time spent on postfiltering, candidate number and maximal number of memorized patterns. Algorithms are sorted in descending order of processing time. The absence of some algorithm in a test means its processing was too long and aborted.

Dataset	Scoring function	Thres. σ_0	Frequent patterns	Freq. LOPs	Algorithm	Total time (s)	Postfilt. ratio	LOP candidates	Max. memo. patterns
Mushrooms	area	0.134	110 K	10	Minimal	1010	99 %	14 K	6.7 K
					Level-wise	36	1 %	11	41 K
8124 data					Maximal	6	5 %	11	40 K
					Lazy	3	10 %	11	20 K
					Lazy+	1.5	26 %	17	17 K
		0.044	4.2 M	19	Maximal	833	0.5 %	19	2 M
					Lazy	220	0.2 %	19	1 M
					Lazy+	74	0.6 %	25	0.8 M
	difference	0.086	328 K	24	Minimal	1950	99 %	27 K	23 K
					Maximal	23	7 %	140	144 K
	edible				Lazy	12	13 %	140	72 K
	minus				Lazy+	6	27 %	140	61 K
	poisonous	0.035	4.7 M	29	Lazy	246	2 %	553	1.1 M
					Lazy+	98	4 %	556	1 M
Breast-cancer	area	0.001	297 K	402	Minimal	93	97 %	28 K	15 K
					Level-wise	76	0.5 %	402	150 K
699 data					Maximal	13	1 %	402	103 K
					Lazy	7	3 %	402	50 K
					Lazy+	3	6 %	430	38 K
	difference	0.001	147 K	383	Minimal	72	98 %	23 K	9 K
	cancerous				Maximal	5	5 %	607	51 K
	minus				Lazy	3	10 %	607	24 K
	healthy				Lazy+	1.6	17 %	622	18 K
Vote	area	0.017	1.5 M	6	Level-wise	585	<0.1 %	7	0.5 M
					Maximal	146	<0.1 %	7	0.7 M
435 data					Lazy	45	0.1 %	42	0.3 M
					Lazy+	24	0.4 %	120	0.2 M
					Minimal	13	36 %	1.3 K	5 K
	difference	0.005	3.3 M	53	Minimal	1000	95 %	304 K	157 K
	republican				Maximal	349	0.2 %	2311	1.4 M
	minus				Lazy	155	0.4 %	2325	0.7 M
	democrat				Lazy+	65	1 %	2415	0.67 M
Chess	area	0.585	328 K	1	Minimal	826	99 %	16 K	31 K
					Level-wise	118	3 %	79	144 K
3196 data					Maximal	26	15 %	79	218 K
					Lazy	14	27 %	79	110 K
					Lazy+	11	35 %	79	110 K
Connect	area	0.907	20 K	0	Minimal	4400	99 %	3300	2.5 K
					Level-wise	290	98 %	214	10 K
67757 data					Maximal	285	99 %	214	14 K
					Lazy	284	99 %	214	7 K
					Lazy+	284	99 %	214	7 K

than GALLOP running time due to the large dataset, differences of speed between GALLOP's versions are completely overwhelmed by the same very long postfiltering. The same reasoning explains why GALLOP-Minimal is the slowest algorithm for all tests but one: the poor filtering capability of the algorithm typically produces a hundred to a thousand more candidates than the other methods so that the very long postfiltering overhelms the very short filtering time. The only exception is the test with the area function and the *Vote* dataset, where GALLOP-Minimal jumps from the last to the first place as show Fig. 9(a) and (c): this surprising result is only possible because *Vote* is a particularly small dataset and that the number of candidates produced by GALLOP-Minimal is relatively small (this is not true anymore for the difference function). However as shown on Fig. 9(c), GALLOP-Minimal only keeps this top position when the number of candidates is kept relatively small, that is, for relatively high frequency threshold.

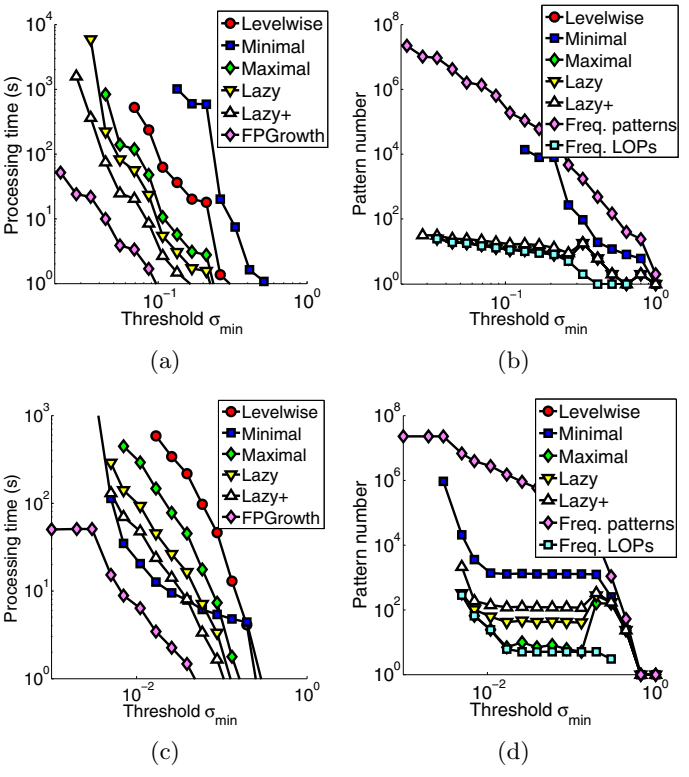


Fig. 9. Processing times for *Mushrooms* (a) and *Vote* (c) using the area function, and candidate numbers bounded by number of frequent patterns and frequent optimal patterns for *Mushrooms* (b) and *Vote* (d). **All axis have a logarithmic scale.** The number of candidates is non-monotonic as a frequent candidate that is dominated by some non-frequent successor M is removed once the frequent threshold decreases and M gets frequent.

5 Conclusions

This article proposes a generic algorithm GALLOP that outperforms with one order of magnitude previous methods to extract locally optimal itemsets, even if long postfiltering might hide this benefit when datasets are very large and many candidates are generated. More fundamentally this article raises a very general problem that is to find patterns optimizing locally any scoring function within the pattern space. Not only some instances of these patterns show interesting properties in the framework of knowledge discovery but optimal patterns along with notions like variation consistency and dominance influence are believed to be part of a more global research perspective: the development of relevant concepts and tools for representing and studying (scoring) functions defined over ordered sets (of patterns).

References

1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Buneman, P., Jajodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, pp. 207–216. ACM Press, New York (1993)
2. Dong, G., Li, J.: Efficient mining of emerging patterns: discovering trends and differences. In: KDD '99: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 43–52. ACM, New York (1999)
3. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Efficient mining of association rules using closed itemset lattices. International Journal of Information Systems 24(1), 25–46 (1999)
4. Boulicaut, J.F., Bykowski, A., Rigotti, C.: Free-sets: A condensed representation of boolean data for the approximation of frequency queries. Data Min. Knowl. Discov. 7(1), 5–22 (2003)
5. Wang, J., Han, J., Lu, Y., Tzvetkov, P.: Tfp: An efficient algorithm for mining top-k frequent closed itemsets. IEEE Trans. Knowl. Data Eng. 17(5), 652–664 (2005)
6. Pennerath, F., Napoli, A.: The model of most informative patterns and its application to knowledge extraction from graph databases. In: Buntine, W. L., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. LNCS, vol. 5782, pp. 205–220. Springer, Heidelberg (2009)
7. Knobbe, A.J., Ho, E.K.Y.: Pattern teams. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 577–584. Springer, Heidelberg (2006)
8. Siebes, A., Vreeken, J., van Leeuwen, M.: Item sets that compress. In: Ghosh, J., Lambert, D., Skillicorn, D.B., Srivastava, J. (eds.) SDM. SIAM, Philadelphia (2006)
9. Bringmann, B., Zimmermann, A.: One in a million: picking the right patterns. Knowl. Inf. Syst. 18(1), 61–81 (2009)
10. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Min. Knowl. Discov. 8(1), 53–87 (2004)