Declarative Heuristic Search for Pattern Set Mining

Tias Guns Siegfried Nijssen Albrecht Zimmermann Luc De Raedt

Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3000 Leuven, Belgium

***Abstract*—Recently, constraint programming has been proposed as a declarative framework for constraint-based pattern mining. In constraint programming, a problem is modelled in terms of constraints and search is done by a general solver. Similar to most pattern mining algorithms, these solvers typically employ exhaustive depth-first search, where constraints are used to prune the search space and make the search viable.**

In this paper we investigate the use of a similar declarative approach to the problem of pattern *set* mining. In pattern set mining one is searching for a small and useful set of patterns.

In contrast to pattern mining, however, exhaustive search is not common in pattern set mining; the search space is often far too large to make such an approach practical. In this paper, we investigate an approach which aims to make general pattern set mining feasible by using a recently developed general solver that supports exhaustive as well as *heuristic* search. The key idea in this solver is that next to a declarative specification of the constraints also a high-level declarative description of the search is given. By separating the model and the search from the solver, the approach offers the advantage of reusing constraints and search strategies declaratively, while also allowing fast heuristic search.

***Keywords*-pattern mining; constraints; constraint programming; greedy search; declarative**

I. Introduction

It is increasingly recognized that pattern mining in itself is not very useful. Most pattern mining algorithms generate large numbers of patterns if an insufficient number of constraints is applied, or if constraint parameters are not tight enough, while they generate only trivial patterns when the constraints are too restrictive. To address this issue, many papers in recent years have studied *pattern set mining* problems [1]; in pattern set mining the aim is to find a small *set* of patterns that is useful with respect to a particular purpose. For instance, when one is interested in building classifiers, a pattern set of interest consists of patterns that jointly discriminate the classes well from each other; when one is interested in a concise description of data, a pattern set that partitions (clusters) the data may be more appropriate; or one could be interested in a small set of diverse patterns instead of many similar ones.

Finding such sets of patterns is a complex and cumbersome task. Many algorithms have been proposed in the last 10 years to address it [1]. Even though a common feature of most of these algorithms is that they perform some kind of greedy or local search, they differ widely in the heuristics and search orders used. Often, the pattern mining step is strictly separated from the pattern set mining step; the constraints used during pattern mining only have superficial relationships to the requirements on the pattern set as a whole. The reason for this approach is that solving the pattern set mining problem at large is usually not possible because the search space is too large to explore using exhaustive search.

Additionally, most algorithms were developed independently and by loosely integrating components. An important factor is the lack of an established framework that enables the easy study and solution of pattern set mining problems as a whole. Current technology makes exploring new problems and complex search strategies complicated and time consuming.

In this paper we investigate the possibilities for studying pattern and pattern set mining in a common declarative framework. Key requirements for this framework are in our opinion the following:

- both the search for patterns and for pattern *sets* should be supported within the same framework, enabling the tight integration between pattern and pattern set mining;
- the framework should support both complete search and heuristic search strategies, to enable efficient search;
- the framework should be declarative both with respect to the task and to the search procedure used, to enable rapid prototyping of new pattern set mining tasks and search strategies.

Ideally, in our vision, a user would be able to develop a range of preferred search strategies as well as a range of useful problems specifications, and would have the capability of mixing these freely; changing from greedy search to complete search for a particular task should not involve changing more than a few lines of code. Implementing entire algorithms for a specific pattern (set) mining task should no longer be necessary.

In recent work we showed that declarative constraint programming methods can be used as a general framework for pattern mining [2]. Constraint programming is a promising approach to declarative pattern mining as it strictly distinguishes the model of a problem from the solver that finds a solution. Within the constraint programming commu-

nity many languages have been developed for declaratively modeling problems, such as Zinc and Essence' [3], [4]; furthermore, many general systems have been developed for solving problems modelled in such a language, such as Gecode and Comet [5], [6]. In recent years, the CP community has extended its declarative approach to also support heuristic search, such as local search and greedy search [6]. Essentially, such systems separate the specification of a problem by means of constraints, from the specification of the search; this allows the same type of search to be used on different problems, and allows the same type of problem to be solved by means of different types of search.

In this paper we select one well-known and groundbreaking representative of such systems, the Comet system [6]. We present a preliminary investigation of its suitability for declaratively modeling and (heuristic) solving of pattern set mining tasks.

The paper is organized as follows. In Section II we present a general overview of the proposed approach; in Section III, we present models of several pattern set mining tasks in terms of constraints; Section IV presents the specification of the search; Section V provides preliminary experimental results and Section VI concludes with a discussion.

II. Constraint Programming & Large Neighbourhood Search

The key idea in constraint programming (CP) [7] is to separate the specification of a problem in terms of constraints from the underlying solver that finds solutions to the specification.

Constraint Specification: In CP a problem is modelled in terms of *constraints* between *variables*. In most solvers, variables should have a *finite* domain, where the domain consists of all values that a variable can take. Typical constraints that can be expressed between variables involve boolean operators as well as basic mathematical operations such as addition and multiplication. An example of a model in terms of constraints is the following:

$$A, B \in \{true, false\} \quad (1)$$
$$X, Y \in \{1, 2, 3, 4\} \quad (2)$$
$$B \leftrightarrow (X + Y \geq 10) \quad (3)$$
$$A \leftrightarrow X \leq 3 \quad (4)$$
$$A \rightarrow B \quad (5)$$

A solution to this model is for instance $A = false$, $B = false$, $X = 4$, $Y = 1$. Equation 1 and 2 specify the domains of variables. Constraints 3 and 4 are known as *reified* constraints, as they tie the value of a boolean variable to the evaluation of a constraint.

In addition, most solvers support the specification of maximization or minimization problems; in this case, in addition to constraints, an optimisation function is specified.

Algorithm 1 Constraint-Search(D)

```
1:  D :=propagate(D)
2:  if D is a false domain then
3:      return
4:  end if
5:  if ∃x ∈ V : |D(x)| > 1 then
6:      Select a variable x with |D(x)| > 1
7:      Select a value d ∈ D(x)
8:      Constraint-Search(D ∪ {x ↦ {d}})
9:      Constraint-Search(D ∪ {x ↦ D(x)\d})
10: else
11:     Output solution
12: end if
```

Constraint Search: Given a specification of a problem in terms of constraints, the task of a CP solver is to search for solutions. Traditional CP solvers use a combination of *propagation* and exhaustive search for this purpose.

Propagation is the process of eliminating possible values from the domains of variables. In our example, for instance, the domains of variables X and Y are such that in constraint (3) the right-hand-side $X+Y < 10$ is true in all cases, hence propagation would eliminate the value $true$ from the domain of variable B. Subsequently, it would be concluded that A needs to be false as well; finally, the variable X would be fixed to value 4. At this point the propagation stops, while Y still does not have a value. The search *branches* by giving a concrete value to Y. After this assignment, propagation can start again. Once a complete solution is found (as in our example), the search can either stop (if one solution is desired) or continue to find alternative solutions.

The general outline of the algorithm is given in Algorithm 1, where $D(x)$ denotes the domain of variable x. Note that one major element of choice is which variable and value to select next during the search. These choices can have an impact on the efficiency of the search.

Some modifications to this approach are usually applied when addressing maximization or minimization problems. In this case the quality of the best solution found so far is stored (f), and the domain of the optimization criterion is maintained as if it were a variable F; search continues as long as the domain of the optimization criterion contains a value that improves upon the best solution found so far.

It is possible that after a number of assignments to variables, the propagation encounters a contradiction (line 2); this is referred to as a *failure*. Usually a CP solver backtracks in this case over its last assignment.

Large Neighbourhood Search: It is clear that the complete search strategy used by traditional CP solvers is not feasible for all constrained optimization problems. An alternative search strategy can be to perform *local search*. In traditional local search, search proceeds by changing one complete solution into a *neighbouring* complete solution; the

Algorithm 2 Large-Neighbourhood-Search(Model M)

1: Create a model M' from M with additional restricting constraints
2: **while** Global stopping criterion is not met **do**
3: Run Constraint-Search on M',
4: till a stopping criterion is met
5: Create a new model M' from M in which
6: 1. new additional restricting constraints are added
7: 2. a constraint $F \geq f$ is included
8: **end while**

neighbouring solution is chosen either greedily or stochastically, taking into account the optimization criterion.

The CP community has endeavoured to combine local search with the traditional mechanisms of CP, which has led to the development of *large neighbourhood search* [8]. The main idea is to make a problem easier to solve by iteratively adding restrictive constraints –such as constraints that fix certain variables– at each step of the local search. Adding constraints to an optimization problem will possibly deteriorate the score of the solutions found, but –if well chosen– simplifies finding solutions. The main idea of the search procedure is given in Algorithm 2, assuming we are solving a maximization problem. Many elements of this algorithm still need to be specified:

- which initial constraints are added?
- when does the iterative loop stop?
- when is the search by the CP solver interrupted?
- based on the results of the CP solver, which constraints are added next and which are no longer applied?

One reason for interrupting the search of the CP solver could be that it encountered too many failed branches. This would likely be the result of a "problematic" variable assignment early on that will however only be corrected after many later assignments are explored. In this case, the order in which the CP solvers considers variables and values can become very influential: a well-specified order may lead to better solutions before the search is interrupted.

Search specification: Given the many choices possible, the main idea of recent CP languages is to augment the explicit constraint model with an equally explicit search specification. One such language is the *Comet* language. We provide examples of this language later on. Within the declarative approach there still is a strict separation between the specification of the task, and the solver that finds a solution. However, there is now the added possibility of specifying the search strategy in a general way.

III. Modeling Pattern and Pattern Set Mining

To apply this general approach to pattern set mining problems, it is necessary to develop specifications of such problems in terms of constraints. For this purpose, we will reuse our earlier work [2], which we briefly summarize here.

A. Pattern Constraints

Let us first consider how to model constraints on individual patterns. We assume given a traditional itemset mining setting, in which we have a binary matrix $\mathcal{D}$, i.e. a matrix in which all elements are either 0 or 1. The rows correspond to elements in a transaction set $\mathcal{T}$ while the columns correspond to elements in a set of items $\mathcal{I}$. The key idea in our work [2] is to formalize a constraint based mining task as a constraint satisfaction problem over a pair of variables (I, T), where I represents an itemset $I \subseteq \mathcal{I}$ and T represents a transaction set $T \subseteq \mathcal{T}$. Both sets are represented by means of boolean variables, i.e. variables I_i and T_t for every item $i \in \mathcal{I}$ and every transaction $t \in \mathcal{T}$. A candidate solution to the constraint satisfaction problem is then one assignment to the variables in $\pi = (I, T)$. For instance, the pattern represented by $\pi = (< 1, 0, 1 >, < 1, 1, 0, 0, 1 >)$ has items 1 and 3, and covers transactions 1, 2 and 5.

Constraints on patterns can now be expressed on these boolean variables as follows.

Coverage: The most important constraint is $coverage(I, T)$, which is defined as follows:

$$\forall t \in \mathcal{T} : T_t = 1 \leftrightarrow \bigwedge_{i \in \mathcal{I}} (\mathcal{D}_{ti} = 1 \vee I_i = 0) \tag{6}$$

Essentially, this constraint enforces that exactly all transactions are selected that contain the selected itemset.

Frequency: The traditional minimum support constraint $minfreq(I, T)$ is defined as follows:

$$\forall i \in \mathcal{I} : I_i = 1 \rightarrow \sum_{t \in \mathcal{T}} T_t \mathcal{D}_{ti} \geq \theta. \tag{7}$$

Note that we use a *reified* constraint here; we showed that this reified constraint is more effectively propagated [2].

By combining coverage and frequency we can express the basic *frequent itemset mining* problem. An example of the syntax for writing down these constraints in the Comet language is given in Figure 1. Lines (14-15) specify the constraint programming variables; lines (18-19) define constants in the model. Lines (15-16) involve bookkeeping, such as reading the data from disk; line (1-12) specify the high-level constraints. We removed the (cumbersome) parameter specifications of the functions from the code here for reasons of readability. Important to note is that these functions essentially enable to specify a problem in terms of the high-level constraints that we discussed earlier. Lines (22-23) use these constraints in the model; line (21) specifies that we are interested in finding all solutions to these constraints; line (25) specifies that a default search order is to be used.

Other constraint-based mining tasks can be equally easily formalized by changing a few lines of this specification. For the closed and accurate mining tasks, we only provide mathematical notation below. They can be specified using syntax similar to the coverage and frequency constraint.

```
 1. function void coverage( parameters ) {
 2.   forall (t in 1..NrT) cp.post(
 3.     Trans[t]==
 4.       (and(i in 1..NrI: !D[t].contains(i))
 5.         !Items[i]));
 6. }
 7. function void minfreq( parameters ) {
 8.   forall (i in 1..NrI) cp.post(
 9.     Items[i]=>
10.       (sum(t in 1..NrT: D[t].contains(i))
11.         Trans[t]>=MinF));
12. }
13.
14. Integer NrI(0);Integer NrT(0);
15. Integer MinF(10);
15. set{int}[] D = readD("filename",NrI,nrT);
16. Solver<CP> cp();
17.
18. var<CP>{bool} Items[1..NrI] (cp);
19. var<CP>{bool} Trans[1..NrT] (cp);
20.
21. solveall<cp> {
22.   coverage(cp,Items,Trans,D,NrI,NrT);
23.   minfreq(cp,Items,Trans,D,NrI,NrT,MinF);
24. }
25. using { label(Items); }
```

Figure 1: Frequent Itemset Mining in Comet

Closedness: An itemset is closed if next to coverage it satisfies the $closed(I,T)$ constraint:

$$\forall i \in \mathcal{I} : I_i = 1 \leftrightarrow \bigwedge_{t \in \mathcal{T}} (\mathcal{D}_{ti} = 1 \vee T_t = 0) \tag{8}$$

Note the similarity to the coverage constraint.

Accuracy: Instead of support, one can also specify a constraint on the accuracy of a pattern. Let $accuracy(T)$ be defined as follows:

$$accuracy(T) = \sum_{t \in \mathcal{T}^+} T_t - \sum_{t \in \mathcal{T}^-} T_t, \tag{9}$$

where we assume that the transactions are grouped into two sets: positive transactions ($\mathcal{T}^+$) and negative transactions ($\mathcal{T}^-$); then one simple way to define the constraint is to require that $accuracy(T) \geq \theta$.

B. Pattern Set Constraints

The setting above can be extended towards $k-$pattern sets, that is, sets of k patterns [9]. We represent a k pattern set by k sets of (I,T) variables, i.e. $\Pi = \{\pi_1, \ldots, \pi_k\}$ where $\forall p = 1, \ldots, k : \pi_p = (I^p, T^p)$. Constraints expressed between these patterns depend on the task at hand.

Here we will limit our study to two tasks, one of which is supervised in nature, the other of which unsupervised.

Concept learning: Concept learning is a supervised pattern set mining task. We assume given a database with two sets of transactions: positive transactions $\mathcal{T}^+$ and negative transactions $\mathcal{T}^-$, and are interested in a set of itemsets that together cover as many positive transactions as possible, while covering only few negative transactions. This can be formalized as the following problem:

$$\begin{array}{rll} \underset{\Pi,T}{\text{maximise}} & accuracy(T), & \\ \text{where} & \forall t \in \mathcal{T} : & T_t = (\vee_{p=1\ldots k} T_t^p), \\ & \forall \pi \in \Pi : & coverage(\pi), \\ & \forall \pi \in \Pi : & closed^+(\pi). \end{array}$$

Note that T_t denotes a boolean variable indicating whether any (at least one) of the patterns covers this transaction.

```
 1. Solver<CP> cp();
 2. var<CP>{bool} Items[1..K,1..NrI](cp);
 3. var<CP>{bool} TranP[1..K,1..NrT_pos](cp);
 4. var<CP>{bool} TranN[1..K,1..NrT_neg](cp);
 5. var<CP>{bool} CoverP[1..NrT_pos](cp);
 6. var<CP>{bool} CoverN[1..NrT_neg](cp);
 7.
 8. maximize<cp>
 9.  (sum(t in 1..NrT_pos) CoverP[t])
10.    - (sum(t in 1..NrT_neg) CoverN[t])
11. subject to {
12.   forall (t in 1..NrT_pos) cp.post(
13.     CoverP[t]==(or(k in 1..K) TranP[k,t]));
14.   forall (t in 1..NrT_neg) cp.post(
15.     CoverN[t]==(or(k in 1..K) TranN[k,t]));
16.   forall (k in 1..K) {
17.     Is=Items[k]; TP=TranP[k]; TN=TranN[k];
18.     coverage(cp,Is,TP,Dpos,NrI,NrT_pos);
19.     coverage(cp,Is,TN,Dneg,NrI,NrT_neg);
20.     closed(cp,Is,TP,Dpos,NrI,NrT_pos);
21.   }
22. }
23. using { label (Items); }
```

Figure 2: Concept learning in Comet

Solving such a problem using exhaustive search can in principle be achieved by the code in Figure 2, which is a simple modification of our earlier code; also note the similarity between this code and our mathematical formulation of the problem. We should note that this solution would not be very efficient due to *symmetries* that are not taken into account: the pattern set $[\pi_1, \pi_2]$ is equivalent to the pattern set $[\pi_2, \pi_1]$. The model in Figure 2 considers both (for reasons of readability); in practice we add an additional constraint to impose a lexicographic ordering on the patterns in the set.

Diverse pattern sets: Many patterns found during a basic pattern mining step have highly similar transaction sets; such patterns can be undesirable. The problem of finding a small, but diverse set of patterns is referred to as the problem of finding *diverse pattern sets*. Many measures can be used to define the similarity between two sets of transactions. One such measure is the *dispersion score* [10]:

$$dispersion(T^i, T^j) = \left(\sum_{t \in \mathcal{T}} (2T_t^i - 1)(2T_t^j - 1) \right)^2. \tag{10}$$

The term $2T_t^i - 1$ essentially transforms a binary {0,1} variable into one of range {-1,1}. An interesting property of this score is that it is both maximal if two patterns cover exactly the same transactions and if one pattern covers exactly the opposite transactions of the other. It can be used to measure the diversity of a pattern set by summing over all pairwise distances:

$$\sum_{i=1}^{k}\sum_{j=1}^{i-1} dispersion(T^i, T^j). \tag{11}$$

Note that this score will always be zero for a pattern set with only one pattern. One possibility to address this problem is to include an additional term

$$\sum_{i=1}^{k} dispersion(T^i, < 1, 1, \ldots, 1 >). \tag{12}$$

For a pattern set with one pattern this score will prefer a pattern that covers approximately half of the transactions.

Instead of dispersion, other scores may be used as well, such as a Jaccard or Dice distance. As long as the function uses basic mathematical operators such as addition and multiplication, any such score can be written down in a CP model and used by the solver.

IV. Modeling Search

A key element in our approach is the specification of search. This section addresses different aspects of search.

A. Modeling Exhaustive Search Order

In the general constraint search algorithm, line 6 of Algorithm 1, several choices can be made with respect to the ordering in which to fix the variables. We will illustrate two possibles orderings here.

Pattern-first order: In this order we first select a pattern and then select each item of that pattern, before selecting the next pattern. The items of a pattern are selected based on their frequency, i.e. the item with the lowest frequency first.

This can be achieved in Comet by replacing the `label(Items)` of line (23) in Figure 2 by the following:

```
using {
  forall ( k in 1..K )
    forall ( i in 1..NrI ) by (freqs_pos[i])
      try<cp> cp.label(Items[k,i],0);
            | cp.diff(Items[k,i],0);
}
```

Essentially, this code specifies in which order the variables have to be considered.

Item-first order: In this order we first choose the item with the lowest frequency, and then select it in each of the patterns in turn. This variable ordering can also easily be achieved with the following Comet code:

```
using {
  forall ( i in 1..NrI ) by (freqs_pos[i])
    forall ( k in 1..K )
      try<cp> cp.label(Items[k,i],0);
            | cp.diff(Items[k,i],0);
}
```

Hence, replacing a few lines of high-level code is enough to change the order in which variables are processed during search.

B. Modeling Large Neighbourhood Search

In this section we consider several heuristic search strategies for pattern set mining and how they can be expressed in the general framework of Section 2. The main idea is to develop alternative strategies for line 1 and 6 of the large neighbourhood search in Algorithm 2.

Greedy Search: The most common search strategy in pattern set mining is *greedy search*. In this search strategy pattern sets are grown iteratively by adding locally optimal patterns to an initially empty pattern set.

Let us consider how we can instantiate this for our two test cases, concept learning and diverse patterns:
Concept learning: we initialize all itemsets in the pattern set to the itemset consisting of all items, i.e. $I_i = \mathcal{I}$. The argumentation is as follows. Let $\Pi' = ((I_1, T_1), (I_2, T_2), \ldots, (I_{k'}, T_{k'}))$ be a set of patterns of size smaller than k and let T_f be the set of transactions covered by the itemset $\mathcal{I}$; then

$$accuracy(T_1 \cup \cdots \cup T_{k'}) = accuracy(T_1 \cup \cdots \cup T_{k'} \cup T_f),$$

as $T_f \subseteq T_i$ for all $1 \leq i \leq k'$. In other words, adding an arbitrary number of T_f itemsets to a pattern set has no effect on its accuracy.

Subsequently, in iterations $p = 1 \ldots k$ we iteratively and temporarily unfix all variables for itemset p only.
Diverse patterns: we initialize all itemsets in the pattern set to the full itemset $\mathcal{I}$. However, during greedy search we let the optimisation function only sum over the patterns already visited. For example, in iteration $s < k$ the optimisation function will sum over patterns 1 to s instead of 1 to k.

Comet code for the concept learning task is illustrated in Figure 3. Line (4) indicates that when the CP solver finishes, the solver should be restarted after executing the lines in the `onRestart` block. Line (5) indicates that the `onRestart` block should also be executed before the first search is performed. Lines (14-15) initialize all patterns as full itemsets; Lines (20-23) fix all patterns, except one, in each iteration; the patterns are fixed to their solutions in previous iterations. Note that in each iteration a complete search is performed to find the best scoring pattern.
Modification Note that variations of Greedy search are easily constructed. For instance, we can create a greedy algorithm which proceeds in iterations, each of which involves finding the first pattern in the pattern set that can be replaced with a pattern that improves the overall quality. Such an algorithm would effectively only grow a pattern set

```
 1. Solver<CP> cp();
 2.
 3. Integer iter(1);
 4. cp.restartOnCompletion();
 5. cp.startWithRestart();
 6. maximize<cp>
 7.    ...  (lines 9-10 of Figure 2)  ...
 8. subject to {
 9.    ...  (lines 12-21 of Figure 2)  ...
10. }
11. using { label(items); }
12. onRestart {
13.   if (iter == 1) {
14.     forall (k in 2..K,i in 1..NrI)
15.       cp.label(Items[k,i],1);
16.   } else {
17.     if (iter > K)
18.       cp.exit();
19.     Solution s = cp.getSolution();
20.     forall (k in 1..K: k!=iter,
21.             i in 1..NrI)
22.       cp.post( Items[k,i] ==
23.         Items[k,i].getSnapshot(s));
24.   }
25.   iter := iter+1;
26. }
```

Figure 3: Greedy concept learning in Comet; only modifications with respect to Figure 2 are shown.

from size k' to size $k'+1$ if none of the patterns π_p with $1 \leq p \leq k'$ can be replaced with a pattern that improves the pattern set's quality.

Large Neighbourhood Search: While the above approach is deterministic, also non-deterministic approaches to pattern set mining have been studied [10]. One such approach iteratively selects a pattern π_p at random from a complete pattern set and searches for a replacement pattern that results in a higher scoring pattern set. We can also achieve such settings within Comet's framework.

An example of such an approach is given in Figure 4. It replaces the `onRestart` block on lines 12-26 in Figure 3. Note that we also need to specify the starting point of the local search. We choose to initialize the search with the best solution found using an exhaustive search that is interrupted after 5000 failures. This is obtained by changing lines (4-5) of Figure 3 to `cp.restartOnFailure(5000)`. Line (7) of the stochastic large neighbourhood search selects an arbitrary pattern for replacement. This search routine will continue endlessly unless a timeout or stopping criterion is added.

Modification Many variations of this stochastic local search strategy can be conceived; for instance, instead of fixing all except one pattern in each iteration, we can also at random select a number of variables among all item variables and search completely over these. Modifications always involve changing the `OnRestart` block of Comet's code.

```
1. onRestart {
2.   Solution s = cp.getSolution();
3.   UniformDistribution dist(1..K);
4.   int not_k = dist.get();
5.   forall (k in 1..K: k!=not_k, i in 1..NrI)
6.     cp.post(Items[k,i] ==
7.       Items[k,i].getSnapshot(s));
8. }
```

Figure 4: Stochastic large neighbourhood search.

Name	Transactions	Items	Density
primary-tumor	336	31	48%
hepatitis	137	68	50%
tic-tac-toe	958	27	33%
audiology	216	148	45%
german-credit	1000	112	34%
australian-credit	653	125	41%
soybean	630	50	32%
lymph	148	68	40%
vote	435	48	33%
heart-cleveland	296	95	47%

Table I: Datasets used in the experiments

V. Experiments

We modelled several of the search strategies described in the previous section in the Comet[1] system. Except for the initialisation when doing greedy (iterative) search, the strategies are independent of the actual constraint specification of the pattern set mining problem. We tested this for the pattern set mining task of concept learning and mining diverse pattern sets. Below, we report on a number of preliminary experiments using different strategies on both tasks. These experiments are mainly meant to demonstrate the flexibility of the declarative approach. The comet system was not designed for pattern mining and is in all likelihood not competitive with specialised mining systems. In this initial study, we restricted ourself to reasonably small datasets. Nevertheless, a number of interesting observations can be made when comparing the different search strategies.

The experiments were performed on PCs running Ubuntu 10.04 with Intel(R) Core(TM)2 Quad CPU Q9550 processors and 4GB of RAM. The datasets were taken from the constraint programming for itemset mining website[2]. The datasets were derived from the UCI Machine Learning repository [11] by discretising numeric attributes into eight equal-frequency bins. To obtain reasonably balanced class sizes we used the majority class as the positive class. Experiments were run on a number of datasets whose basic properties are listed in Table I.

A. Exhaustive pattern set mining

We start by checking the feasibility of exhaustive pattern set mining for the concept learning task, as well as the influence of different (exhaustive) search strategies. The

[1]version 2.1.1, from http://www.comet-online.org/Downloads.html
[2]http://dtai.cs.kuleuven.be/CP4IM/datasets/

k	Item-first	Circular	Pattern-first
2	813 s	840 s	1172 s
3	1536 s	1549 s	1640 s
4	1800 s	1800 s	1800 s
5	1800 s	1800 s	1800 s
6	1800 s	1800 s	1800 s

Table II: Concept learning with set size k, average runtime over the datasets in Table I, timeout of 1800 seconds.

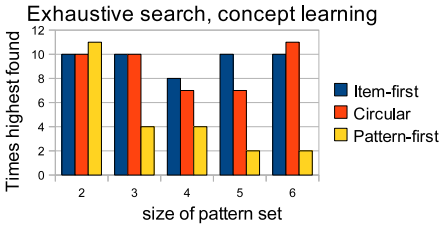

Figure 5: Nr of times the highest scoring pattern set is found by that strategy, on each of the 11 datasets in Table I.

strategies used are *Pattern-first* order, *Item-first* order and *Circular*. The latter selects the least frequent item, strictly among each pattern in turn. Potential ties in the ordering are randomly broken.

Table II shows that for a set size of 4 or higher, none of the exhaustive strategies was able to prove within 30 minutes that it found the optimal solution for any of the datasets. The pattern-first ordering seems to be somewhat slower than the other two. Figure 5 shows how many times the strategies found the pattern set with the highest score compared to the other strategies. The pattern-first ordering performs significantly worse for set sizes above 2. The *item-first* and *circular* strategies are close to each other. We consider this evidence that choosing the right search strategy is not straight-forward and having the opportunity to explore different ones can aid the mining process.

In the rest of the experiments, we use the item-first strategy as the exhaustive search strategy. Having a good exhaustive search strategy is important as exhaustive search is done between every restart.

B. Heuristic pattern set mining

We perform a number of experiments on two pattern set mining tasks, namely concept learning and mining a diverse pattern set. We use the search strategies discussed in Section IV-B: *exhaustive search* (All), *greedy search* (Gr), *greedy iterative search* (Gr-it), *per pattern Large Neighbourhood Search* (LNS-p) and *random LNS* (LNS-r). Although these experiments are preliminary, we report on a number of interesting observations.

Concept learning: Table III shows win/draw/loss counts between a number of search strategies. The exhaustive strategy is very inefficient and hence a more naive greedy approach already performs much better (Gr VS All). The iterative greedy procedure is only able to find a better solution on one dataset, and does not reach the same solution

	Gr VS All	Gr-it VS Gr	LNS-r VS LNS-p	LNS-r VS Gr
prim.-tumor	3/0/1	0/4/0	0/0/4	3/0/1
hepatitis	3/0/1	0/4/0	2/0/1	4/0/0
tic-tac-toe	2/0/2	0/4/0	2/1/1	4/0/0
audiology	4/0/0	0/4/0	2/1/1	4/0/0
germ.-credit	4/0/0	0/4/0	4/0/0	4/0/0
austr.-credit	4/0/0	0/2/2	4/0/0	2/0/2
soybean	3/1/0	0/4/0	0/1/3	1/0/3
lymph	3/0/1	4/0/0	1/0/3	2/0/2
vote	4/0/0	0/4/0	1/0/3	1/0/3
heart-clev.	4/0/0	0/4/0	4/0/0	3/0/1
total	34/1/5	2/34/2	21/3/16	26/0/14

Table III: win/draw/loss count of different search strategies for the concept learning task. For each dataset, pattern sets of size 3, 6, 9 and 12 were searched for. For LNS, average results over 5 runs were used.

within the timeout on another dataset (Gr-it VS Gr). There is hence no clear benefit to using this strategy for this task.

LNS-r performs slightly better than LNS-p. Additionally, random LNS is often better than a greedy approach. This can be attributed to the fact that the greedy and per-pattern LNS strategies perform (exhaustive) search for single patterns each iteration. Hence, per-pattern strategies can get stuck in a local optimum determined by the other patterns. The random LNS strategy on the other hand is less likely to get stuck in a local optimum.

Dispersion score: The dispersion score has a number of interesting properties: there is no known pruning strategy to use nor is there an intuitive way to order variables during the search. Additionally, finding the most dispersed pattern to add to a set becomes harder as the set grows. Current approaches use post-processing of all (frequent) patterns or greedily adding patterns using a random walk strategy [10].

In our experiments exhaustive search performs very poorly. Only on 3 datasets a solution is found for a set size as low as 3. Greedy search performs better, but for a set size of 12 it no longer finds a solution on four of the eleven datasets. At this set size, exhaustively searching for the best pattern to add to the set becomes prohibitive. Both LNS strategies always find a solution. Table IV shows that random LNS does better than per-pattern LNS (LNS-r VS LNS-p) and greedy search (LNS-r VS Gr). Random LNS performs a kind of random walk; hence this result concurs with the random walk strategy proposed in the literature. Perhaps a combination of random walks and greedy search is even more promising.

VI. Discussion

Our aim in this work was to simplify the study of pattern set mining tasks and search strategies by putting these into a common declarative framework. Based on our preliminary investigations, can we now conclude that the *Comet* framework developed in the constraint programming community is suitable for this purpose?

Within the limited scope of this study, we believe the attempt was mostly successful: we showed that the same constraint specification can be used with a specification of

	Gr VS All	Gr-it VS Gr	LNS-r VS LNS-p	LNS-r VS Gr
prim.-tumor	3/0/1	3/1/0	4/0/0	4/0/0
hepatitis	4/0/0	3/1/0	4/0/0	2/1/1
tic-tac-toe	2/1/0	1/3/0	0/4/0	1/3/0
audiology	3/0/0	2/1/0	4/0/0	1/0/3
germ.-credit	3/0/0	2/1/0	4/0/0	3/0/1
austr.-credit	3/0/0	1/1/1	4/0/0	1/0/3
soybean	3/0/0	2/0/1	3/0/1	4/0/0
lymph	3/0/0	2/0/1	4/0/0	2/0/2
vote	3/0/1	4/0/0	4/0/0	4/0/0
heart-clev.	4/0/0	0/1/0	4/0/0	3/0/1
total	31/1/3	20/9/3	35/4/1	25/4/11

Table IV: win/draw/loss count of different search strategies for the dispersion score. For each dataset, pattern sets of size 3, 6, 9 and 12 were searched for. For LNS, average results over 5 runs were used.

exhaustive search, greedy search and large neighbourhood search methods. Furthermore, the search specifications could be reused –with some modifications– across different tasks, as we demonstrated for the task of concept learning and mining a high dispersion pattern set. For many settings, the developed models are arguably also elegant and concise, as demonstrated by the code included in this paper.

At the same time, there is however also room for improvement in the modeling language. Even though the search specification in the Comet language is already high-level and declarative compared to most imperative languages, in some cases we found that implementing a search strategy still requires quite low level considerations: to mention one example, Comet makes distinctions between call-by-value and call-by-reference, hence still requiring users to consider memory management. Furthermore, to implement greedy search strategies we could not literally reuse the search specification, as minor modifications were needed in the code that prepares the next round of pattern search. Ideally, we could separate this part of the code from the overall search as well. Nevertheless, one can argue that these disadvantages are minor compared to the advantages that the current system already provides.

We used the framework to perform a number of initial experiments in which we compared the different search strategies on the two pattern set mining tasks. We observed that different search strategies are beneficial for different tasks, and that using random Large Neighbourhood Search performed remarkably well. This suggests that the use of random walks in pattern set mining should perhaps be studied in more detail.

A large number of open questions remain for a more extended study:

- We limited our focus to techniques that use an exhaustive constraint solver at some point. Within Comet *pure* local search is also supported, i.e. local search without using the basic propagation principles of CP. The use of these primitives should also be investigated.
- The limited number of pattern set mining tasks studied in this paper could be solved using multiple search strategies with only minor modifications to the specification of the search. It remains to be seen to how many other tasks this observation extends. Similarly, it remains to be seen how many pattern set mining tasks can be modelled in terms of constraints, such as learning decision lists.
- We limited this study to rather small datasets. We encountered some scalability issues when trying to apply it on larger datasets. It remains a question for further study whether extensions of the solver with data structures that support data mining better are needed.
- We limited the order in which variables are tried during the exhaustive search to deterministic orders. As random LNS performs very well on the pattern set mining task, it should be considered to randomize the order of variables during exhaustive search as well.

Finally, we restricted this study to pattern set mining. We believe there is ample opportunity for general declarative tools for data mining and machine learning at large.

Acknowledgements. This work was supported by a Postdoc and project "Principles of Patternset Mining" from the Research Foundation—Flanders, as well as a grant from the Agency for Innovation by Science and Technology in Flanders (IWT-Vlaanderen).

REFERENCES

[1] B. Bringmann, S. Nijssen, N. Tatti, J. Vreeken, and A. Zimmermann, "ECML/PKDD tutorial on mining sets of patterns," 2010.

[2] T. Guns, S. Nijssen, and L. D. Raedt, "Itemset mining: A constraint programming perspective," *Artificial Intelligence*, vol. 175, no. 12-13, pp. 1951 – 1983, 2011.

[3] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace, "The design of the zinc modelling language," *Constraints*, vol. 13, no. 3, pp. 229–267, 2008.

[4] A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernández, and I. Miguel, "Essence : A constraint language for specifying combinatorial problems," *Constraints*, vol. 13, no. 3, pp. 268–306, 2008.

[5] Gecode Team, "Gecode: Generic constraint development environment," 2010. [Online]. Available: http://gecode.org

[6] P. V. Hentenryck and L. Michel, *Constraint-Based Local Search*. The MIT Press, 2005.

[7] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming*. Elsevier, 2006.

[8] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science. Springer, 1998, vol. 1520, pp. 417–431.

[9] T. Guns, S. Nijssen, and L. De Raedt, "k-pattern set mining under constraints," *IEEE Transactions on Knowledge and Data Engineering*, vol. To Appear, 2011, also as Technical Report CW596, Oct 2010.

[10] U. Rückert and S. Kramer, "Optimizing feature sets for structured data," in *ECML*, ser. Lecture Notes in Computer Science, vol. 4701. Springer, 2007, pp. 716–723.

[11] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml