*Article*

# Computational Complexity and ILP Models for Pattern Problems in the Logical Analysis of Data

**Giuseppe Lancia *,† and Paolo Serafini †**

Department of Mathematics, Computer Science and Physics, University of Udine, 33100 Udine, Italy; paolo.serafini@uniud.it
* Correspondence: giuseppe.lancia@uniud.it; Tel.: +39-0432-558454
† These authors contributed equally to this work.

**Abstract:** Logical Analysis of Data is a procedure aimed at identifying relevant features in data sets with both positive and negative samples. The goal is to build Boolean formulas, represented by strings over {0,1,-} called *patterns*, which can be used to classify new samples as positive or negative. Since a data set can be explained in alternative ways, many computational problems arise related to the choice of a particular set of patterns. In this paper we study the computational complexity of several of these pattern problems (showing that they are, in general, computationally hard) and we propose some integer programming models that appear to be effective. We describe an ILP model for finding the minimum-size set of patterns explaining a given set of samples and another one for the problem of determining whether two sets of patterns are equivalent, i.e., they explain exactly the same samples. We base our first model on a polynomial procedure that computes all patterns compatible with a given set of samples. Computational experiments substantiate the effectiveness of our models on fairly large instances. Finally, we conjecture that the existence of an effective ILP model for finding a minimum-size set of patterns equivalent to a given set of patterns is unlikely, due to the problem being NP-hard and co-NP-hard at the same time.

**Keywords:** logical analysis of data; pattern problems; data mining; computational complexity; integer linear programming models

check for updates

**Citation:** Lancia, G.; Serafini, P. Computational Complexity and ILP Models for Pattern Problems in the Logical Analysis of Data. *Algorithms* **2021**, *14*, 235. https://doi.org/10.3390/a14080235

Academic Editor: Francisco Saldanha da Gama

Received: 7 July 2021
Accepted: 6 August 2021
Published: 9 August 2021

## 1. Introduction

One of the main consequences of the constant progress of technology together with the massive use of computers in many aspects of our lives has been the creation of large repositories of data storing information of all sorts. A major problem related to these huge data sets is the one of discovering relevant patterns that separate the noise from important information and of deriving rules for clustering the data into classes sharing essential common features. To this aim, the fields of study known as *data mining* [1,2] and *feature selection* [3–5] have recently emerged among the most relevant applications of modern computer science.

In this paper we focus on some mathematical issues that arise from data mining problems. A very common situation for data mining problems is to represent the starting information by a two-dimensional array, in which the rows correspond to *samples* (or *individuals*) while the columns correspond to their characteristics (also called *features*).

If the features are Boolean, one of the tools that can be used to extract interesting information is the so-called Logical Analysis of Data (LAD [6–8]). Consider for instance a data set consisting of a binary matrix of $m$ rows and $n$ columns, in which some rows are labeled as *positive* while the remaining rows are labeled as *negative* (for instance, in the case of a molecular biology experiment using a device called "microarray" which measures the level of gene expression in cells, the values 0 and 1 would be related to the level being, respectively, "normal" or "abnormal" [9–11]).

The Logical Analysis of Data has the objective of discovering a set of simple Boolean formulas (or "rules") that can be used to classify new binary vectors $(b_1, \dots, b_n)$. Each rule describes what the value of some bits must be for a vector to be classified as positive or negative. For instance, a "positive rule" could be

$$(b_2 = \texttt{1}) \wedge (b_5 = \texttt{0}) \wedge (b_9 = \texttt{0})$$

meaning that any vector with a 1 in the 2nd component, and a 0 in the 5th and 9th component is classified as positive. Similarly, there can be some "negative rules" which specify which vectors should be classified as negative.

A rule such as the above can be conveniently represented by a *pattern*, which is a string over the alphabet $\{\texttt{0}, \texttt{1}, \texttt{-}\}$. The characters `0` and `1` in the pattern specify which positions must be matched exactly by a binary vector to satisfy the rule, while the character `-` is a wildcard that can be indifferently matched by either `0` or `1`. In particular, if $n = 10$ the pattern corresponding to the above rule would be

```
-1--0---0-
```

If $r$ is a rule and $p$ is the pattern corresponding to $r$, then a binary vector $b$ satisfies the rule if and only if

$$\bigwedge_{k: p_k \in \{0,1\}} (b_k = p_k)$$

We say that the pattern $p$ *covers* all vectors $b$ for which the above holds. In view of the equivalence of rules and patterns, we can talk of positive/negative patterns in place of positive/negative rules.

The objective of LAD is to infer positive and negative patterns from the data in such a way that (i) each positive row is covered by at least one of the positive patterns, while no negative row is and (ii) each negative row is covered by at least one of the negative patterns, while no positive row is. This approach has been successfully applied to many contexts in both bioinformatics and biomedicine [8].

Since there might be many alternative sets of patterns explaining a given instance of LAD, one has to introduce a suitable criterion for choosing a specific solution. In particular, an *Okkam's razor* strategy would suggest seeking the simplest possible solutions, i.e., the sets with a minimum number of patterns. Finding a min-size set of patterns which cover a given set of vectors is called the *Pattern Cover Minimality* problem.

Other problems arising from the analysis of patterns are related to understanding whether two different sets of rules actually explain the same data set, or, in other words, the two pattern sets are equivalent. In particular we would like also to know whether a given set of rules explains all possible data, and so is in some sense "useless". On the opposite side we would like to know whether there are some data that cannot be explained by a particular set of rules.

In addition, given a set of patterns we would like to know whether there exists another smaller set of patterns that explains the same data set. This problem that we call *Pattern Equivalence Minimality* looks similar to Pattern Cover Minimality. The difference is that here we start from a pattern set and not from a data set. Though patterns can be expanded into strings and we might solve a Pattern Cover Minimality problem from these strings, it is obviously computationally intractable expanding the patterns. Hence we should be able to find a better set of patterns starting directly from the given pattern set.

In the following we will review the computational complexity of these pattern problems, which are, in general, quite complex [12] (see also [13] for a fixed-parameter analysis of some related pattern problems). We then give an integer linear programming (ILP) formulation for Pattern Cover Minimality and for Pattern Equivalence and we address the effectiveness of our formulations by means of extensive computational experiments. An ILP formulation for Pattern Cover Minimality is also given in Boccia et al. [14]. The formulation we propose in this paper reduces the problem to a Set Covering with a (lowdegree) polynomial number of columns. Pattern Cover models for non-binary data are, e.g., a branch-and-price procedure described in [15] and heuristic procedures proposed in [16].

Formulating a solution procedure for Pattern Equivalence Minimality seems quite challenging since, as we will prove in the paper, this problem is NP-hard and co-NP-hard at the same time.

The paper is organized as follows. In Section 2 we provide precise mathematical definitions of the concept we are dealing with and the related problems. In Section 3 we investigate the computational complexity of the problems defined in the previous section. In Section 4 we investigate strings and patterns that are mutually compatible. In particular, we provide a polynomial algorithm to list all patterns that are compatible with a given set of strings (the inverse problem of listing all strings compatible with a set of patterns is necessarily exponential). In Section 5 we give ILP models both for the Pattern Cover Minimality and Pattern Equivalence problems. These procedures are tested in Section 6 devoted to the computational experiments. Some conclusions are drawn in Section 7.

## 2. Basic Definitions

A *binary string* (or, simply, a *string*) $s$ is a sequence of symbols where each symbol can be either `0` or `1`. With *n-binary string* we denote a binary string of length $n$. An *n-pattern* (or simply a *pattern* $p$ is sequence of symbols where each symbol can take the values `0`, `1` or `-`, i.e.,

$$s_i \in \{\mathtt{0},\mathtt{1}\}, \qquad p_i \in \{\mathtt{0},\mathtt{1},\mathtt{-}\}, \qquad i = 1,\dots n.$$

We call the symbol `-` a *gap*. With *n-pattern* we denote a pattern of length $n$. Notice that a string is in fact a particular pattern, i.e., a pattern without gaps. A pattern $p$ *covers*, or *generates*, a string $s = (s_1 \cdots s_n)$ if $s_k = p_k$ for each $k$ such that $p_k \in \{\mathtt{0},\mathtt{1}\}$. The *span* of $p$ is the set of all binary strings generated by the pattern $p$, i.e.,

$$S(p) = \left\{s \in \{\mathtt{0},\mathtt{1}\}^n : s_i = p_i \text{ if } p_i \in \{\mathtt{0},\mathtt{1}\}\right\}$$

Given a set $P$ of patterns the span $S(P)$ of $P$ is the set

$$S(P) = \bigcup_{p \in P} S(p)$$

Two sets $P$ and $Q$ of patterns are *equivalent* if $S(P) = S(Q)$. A set of patterns $P$ is a *minimum* set if $|P| \leq |Q|$ for each set of patterns $Q$ equivalent to $P$.

We say that a pattern $p$ is *compatible* with a set of strings $S$ if $S(p) \subseteq S$. Similarly we say that a string $s$ is compatible for a pattern $p$ if $s \in S(p)$. We denote by $P(S)$ the set of all compatible patterns for $S$. Let $p$ be a pattern compatible with a given set $S$ of strings. If there is no compatible pattern $p'$ such that $S(p) \subset S(p')$, we say that $p$ is *maximal* (for $S$). We denote by $P^*(S)$ the subset of maximal patterns in $P(S)$. Given a set $S$ of strings and a set $P$ of patterns we say that $P$ is compatible for $S$ if each $p \in P$ is compatible for $S$ and so $P \subseteq P(S)$ and $S(P) \subseteq S$.

Moreover, a set of patterns $P$ is called a *cover* of $S$ if $S(P) = S$. Notice that all covers of $S$ are equivalent to each other and $S$, viewed as a set of patterns, is equivalent to each of its covers.

A set $P$ of patterns is said to be *complete* if $S(P) = \{\mathtt{0},\mathtt{1}\}^n$, i.e., if $P$ generates all possible $n$-binary strings. Clearly, $\{(\mathtt{-}\,\mathtt{-}\cdots\mathtt{-})\}$ is trivially complete.

We note that $S$ is a set and so it does not contain duplicate strings. We assume this to be true also when we represent a set of $m$ strings of length $n$ as an $m \times n$ array of zeros and ones.

## 3. Computational Complexity Results

The previous definitions lead to the following decision problems [12]:

1. PATTERN COVER: given a set $S$ of strings and a set $P$ of patterns, is $P$ a cover of $S$, i.e., $S = S(P)$?
2. PATTERN COVER MINIMALITY: given a set $S$ of strings and a constant $K$, does there exist a cover $P$ of $S$ such that $|P| \leq K$?
3. PATTERN EQUIVALENCE: given two sets $P$, $Q$ of patterns, are they equivalent, i.e., $S(P) = S(Q)$?
4. PATTERN EQUIVALENCE MINIMALITY: given a set $P$ of patterns and a constant $K < |P|$, does there exist an equivalent set of patterns $Q$ such that $|Q| \leq K$?
5. PATTERN COMPLETENESS: given a set $P$ of patterns, is it complete, i.e., $S(P) = \{0, 1\}^n$?
6. PATTERN INCOMPLETENESS: given a set $P$ of patterns, is it not complete, i.e., does there exist a string $s \in \{0, 1\}^n$ such that $s \notin S(P)$?

We first note that, given a set $P$ of patterns and a string $s \in \{\texttt{0}, \texttt{1}\}^n$, determining whether $s \in S(P)$ or $s \notin S(P)$ is polynomial. Indeed, given a pattern $p$ and a string $s$ we may check in time $O(n)$ whether $s$ can be generated by $p$ or not. Hence, given a set $P$ of patterns we have to repeat the check for each $p \in P$. If the check is false for each $p \in P$ we have $s \notin S(P)$, otherwise we have $s \in S(P)$.

**Proposition 1.** *PATTERN COVER is polynomial.*

**Proof.** For each $s \in S$ we check whether $s \in S(P)$ or not. Hence in time $O(n\,|S|\,|P|)$ we may decide whether $S \subseteq S(P)$ or not. In order to decide whether $S(P) \subseteq S$ or not, for each pattern $p \in P$ we count the number $n(p)$ of strings in $S$ compatible for $p$. Let $k$ be the number of gaps in $p$. Then $p$ is compatible for $S$, i.e., $S(p) \subseteq S$, if and only if $n(p) = 2^k$. Computing $n(p)$ can be done in time $O(n\,|S|)$. Overall, checking whether $S(P) \subseteq S$ takes $O(n\,|S|\,|P|)$ time. □

**Proposition 2.** *PATTERN COVER MINIMALITY is NP-complete.*

**Proof.** We observe that PATTERN COVER MINIMALITY is basically the same as MINIMUM DISJUNCTIVE NORMAL FORM (see [17] p. 261), which we repeat here for the sake of completeness: Given a set $U = \{u_1, u_2, \ldots, u_n\}$ of variables, a set $A \subseteq \{T, F\}^n$ of "truth assigments", and an integer $K > 0$, does there exist a disjunctive normal form expression $E$ over $U$, having no more than $K$ disjuncts, which is true for precisely the assignments in $A$ and no others?

We show the equivalence of the two problems by the following map which builds a PATTERN COVER MINIMALITY instance. Each element $a \in A$ becomes an input binary string (with 0 representing false, and 1 representing true), while each disjunct $d$ is mapped into a pattern $p$ such that $p_i = \texttt{1}$ if $u_i$ appears in $d$, $p_i = \texttt{0}$ if $\neg u_i$ appears in $d$ and a $p_i = \texttt{-}$ otherwise. □

**Proposition 3.** *PATTERN INCOMPLETENESS is NP-complete.*

**Proof.** We reduce SAT to PATTERN INCOMPLETENESS. Given a SAT instance with $n$ literals and $m$ clauses we derive a set $P$ of $m$ patterns $p^k$, $k = 1, \ldots, m$, (each pattern associated to each clause), as follows: for each variable $i$ and each clause $k$, if the literal $x_i$ is present in the clause, we set $p_i^k = \texttt{0}$, if the literal $\neg x_i$ is present in the clause, we set $p_i^k = \texttt{1}$, and if neither $x_i$ nor $\neg x_i$ are present in the clause, we set $p_i^k = \texttt{-}$ (note that the pattern values $p_i^k \in \{0, 1\}$ are set opposite to the truth values of the literals $x_i$).

Assume SAT is satisfiable and let $x$ be a satisfying truth assignment. Define a string $s$ as $s_i = \texttt{1}$ if $x_i =$ TRUE and $s_i = \texttt{0}$ if $x_i =$ FALSE. By assumption, at least one literal of each clause must be true, and so for each $p^k \in P$ at least one of the symbols $s_i$ corresponding to 0, 1 positions of $p^k$ must be different from $p_i^k$, due to the particular construction of $p^k$.

It follows that $s$ cannot be in $S(p^k)$ for all $k$ and so $s$ cannot be in $S(P)$. In a similar way, given a string $s$ not in $S(P)$ we can reverse the previous reasoning and obtain a satisfying truth assignment for the SAT instance.

To see that PATTERN INCOMPLETENESS is also in NP it suffices to observe that verifying that a string $s \notin S(P)$ does not belong to $S(P)$ takes polynomial time, as previously described. □

Since PATTERN INCOMPLETENESS and PATTERN COMPLETENESS are complements of each other, we have:

**Corollary 1.** *PATTERN COMPLETENESS is co-NP-complete.*

**Proposition 4.** *PATTERN EQUIVALENCE is co-NP-complete.*

**Proof.** We transform PATTERN COMPLETENESS into PATTERN EQUIVALENCE. Given a set $P$ of patterns, instance of PATTERN COMPLETENESS, the corresponding instance of PATTERN EQUIVALENCE consists of the set $P$ plus a set $Q$ containing only the pattern $(\texttt{-}\texttt{-}\cdots\texttt{-})$ (which generates $\{\texttt{0},\texttt{1}\}^n$). For a no-instance, there exists a string $s \in S(P)$ and $s \notin S(Q)$, or vice versa, and this $s$ is a short certificate. □

**Proposition 5.** *PATTERN EQUIVALENCE MINIMALITY is co-NP-hard.*

**Proof.** We describe a transformation from PATTERN COMPLETENESS. Given an instance of PATTERN COMPLETENESS we define a corresponding instance of PATTERN EQUIVALENCE MINIMALITY by choosing $K = 1$. Without loss of generality, we may assume that for each $i$, the values $p_i$'s across all $p \in P$ are not all identical (since, otherwise, we may discard each position where all symbols are equal and reduce the instance to an equivalent one). At this point, the only pattern that can be equivalent to $P$ is $(\texttt{-}\texttt{-}\ldots\texttt{-})$. □

Since PATTERN COVER MINIMALITY is a particular case of PATTERN EQUIVALENCE MINIMALITY we have by Proposition 2:

**Proposition 6.** *PATTERN EQUIVALENCE MINIMALITY is NP-hard.*

By Propositions 5 and 6, PATTERN EQUIVALENCE MINIMALITY is both NP-hard and co-NP-hard. To date it is not known whether the classes of NP-complete problems and co-NP-complete problems coincide or are disjoint. The widely believed conjecture is that they are disjoint. Following this conjecture we conclude that it is unlikely that PATTERN EQUIVALENCE MINIMALITY is in NP or in co-NP, and so we expect its complexity to be beyond the classes NP and co-NP.

It is obvious that, given a set $P$ of patterns, generating $S(P)$ takes exponential time in general for the mere fact that $|S(P)|$ can be of exponential size. It is perhaps surprising that the reverse, i.e., given a set of strings $S$, generating $P(S)$ is polynomial. Indeed it turns out that $P(S)$ is of polynomial size and also the algorithm that generates $P(S)$ is polynomial. We devote the next section to this issue.

## 4. Compatible Patterns

We describe a procedure to compute $P(S)$, that is the set of all compatible patterns for a set $S$ of strings. The analysis of this procedure shows that the number of compatible patterns is polynomial ($\leq O(|S|^{\log_2 3})$).

We define a recursion that produces a set $\mathcal{P}(S)$ of patterns and we will show that $\mathcal{P}(S) = P(S)$. We assume there are no duplicates in $S$. The recursive calls create string sets that satisfy this property. The length of each string in a generic set $R$ of strings (clearly all of equal length) is denoted by $n(R)$. For a generic set $R$ and $1 \leq c \leq n(R)$, $S(R, c)$ is the set of strings of length $(n - c + 1)$ obtained from $R$ by taking, for each $s \in R$, only the elements $s_c, s_{c+1}, \dots, s_{n(R)}$. Furthermore, for $s$ a string and $X = \{x_1, \dots, x_k\}$ a set of strings, we denote by

$$s \circ X = \{sx_1, \dots, sx_k\}$$

the set obtained by appending $s$ as a prefix to all strings in $X$.

The recursive algorithm to compute $\mathcal{P}(S)$ consists of:

• base case: **if** $|S| = 1$ then **return**$(S)$ (the entire $S$, seen as a single string);

• recursion: given an input set $R$ of strings let $c \le n(R)$ be the first index such that there are two strings in $R$ whose $c$-th elements are different (if there is not such an index all strings in $R$ would be identical, contradicting the hypothesis of no duplicates). Hence all prefixes $s_1, \dots, s_{c-1}$ are equal for each $s \in R$. Let $\bar{s}$ be this common prefix. Let $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_*$ be defined as follows:

1. Let $R_0 = \{s \in R : s_c = \mathtt{0}\}$ and $S_0 := S(R_0, c+1)$. Then $\mathcal{P}_0 := \mathcal{P}(S_0)$ (recursive call).
2. Let $R_1 = \{s \in R : s_c = \mathtt{1}\}$ and $S_1 := S(R_1, c+1)$. Then $\mathcal{P}_1 := \mathcal{P}(S_1)$ (recursive call).
3. Let $R_*$ be the set of all strings $s \in R_0$ for which there exists $s' \in R_1$ such that $s_i = s'_i$ for all $i > c$ (note that $s$ and $s'$ differ only at the $c$-th element and that the strings $s$ and $s'$ (if any) go in pairs due to the hypothesis of no duplicates) and let $S_* := S(R_*, c+1)$ (recursive call).
   **If** $R_* = \emptyset$ **then** $\mathcal{P}_* := \emptyset$ **else** $\mathcal{P}_* := \mathcal{P}(S_*)$.

   Then, **return**$(s \circ ((\mathtt{0} \circ \mathcal{P}_0) \cup (\mathtt{1} \circ \mathcal{P}_1) \cup (\mathtt{-} \circ \mathcal{P}_*)))$

**Theorem 1.** *Let $S$ be a string set and let $\mathcal{P}(S)$ be the set produced by the recursive algorithm on $S$. Then $P(S) = \mathcal{P}(S)$.*

**Proof.** We use induction on $|S|$. Base case: If $|S| = 1$ the assert is clearly true.

Inductive step: Assume $|S| > 1$ and the theorem holds for all sets with $|S| - 1$ strings. With the notation of the algorithm,

$$\mathcal{P}(S) = s \circ ((\mathtt{0} \circ \mathcal{P}_0) \cup (\mathtt{1} \circ \mathcal{P}_1) \cup (\mathtt{-} \circ \mathcal{P}_*)))$$

Let $p \in P(S)$. Since $s$ (possibly null) is a prefix of each string in $S$, $p$ must start with $s$ as well. Assume $p = sxq$ with $x \in \{\mathtt{0}, \mathtt{1}, \mathtt{-}\}$. If $x = \mathtt{0}$ then $q$ is a pattern compatible for $S_0$ and, by induction, $q \in \mathcal{P}(S_0)$. Therefore, $p \in s \circ (\mathtt{0} \circ \mathcal{P}(S_0))$, so that $p \in \mathcal{P}(S)$. A similar argument shows that also if $x = \mathtt{1}$ it is $p \in \mathcal{P}(S)$. Now, if $x = \mathtt{-}$, then for each string $a$ generated by $q$, the pattern $p$ generates both $s\mathtt{0}a$ and $s\mathtt{1}a$. Therefore, $a$ had to be a string both in $S_0$ and in $S_1$. This means that $q$ had to be a pattern compatible for $S_*$. Since, by induction, all such patterns are in $\mathcal{P}_*$ then $p \in s \circ (\mathtt{-} \circ \mathcal{P}_*)$ so that $p \in \mathcal{P}(A)$.

Now, assume $p \in \mathcal{P}(S)$. Let $p = sxq$, with $x \in \{\mathtt{0}, \mathtt{1}, \mathtt{-}\}$. If $x \in \{\mathtt{0}, \mathtt{1}\}$, let $sxy$ be a string generated by $p$. Then $y \in R_0 \cup R_1$ and therefore $sxy \in R$, so $p \in P(S)$. If $x = \mathtt{-}$ let $s\mathtt{0}y$ be a string generated by $p$. Then also $s\mathtt{1}y$ is generated by $p$. Then $y \in R_0 \cap R_1$ so that $s\mathtt{0}y, s\mathtt{1}y \in R$ and $p \in P(S)$. □

**Theorem 2.** $|P(S)| \le |S|^{\log_2 3}$.

**Proof.** Let $T(m)$ be an upper bound to the cardinality of $P(S)$ for a set $S$ with $m = |S|$. From the previous theorem and from the algorithm recursive calls we see that

$$T(m) = \max\{T(m-k) + 2\,T(k) : k = 1, \dots, \lfloor m/2 \rfloor\}, \qquad T(1) = 1 \tag{1}$$

By applying the Master Theorem (see p. 66 [18]) we get

$$T(m) \le m^{\log_2 3}$$

□

These are the first values of $T(m)$ for $m = 1, \ldots, 16$, according to (1):

$$T(m) = 1, 3, 5, 9, 11, 15, 19, 27, 29, 33, 37, 45, 49, 57, 65, 81 \ldots \tag{2}$$

This the sequence A006046 as listed in the On-line Encyclopedia of Integer Sequences (OEIS) and our derivation seems to add a new meaning to that sequence. Note from (2) that in particular for $m = 2^k$ we have $T(m) = 3^k = m^{\log_2 3}$. Clearly, if $S$ consists of all strings of length $k$, so $|S| = 2^k$, then $|P(S)| = 3^k$. So the bound $T(m)$ for $|P(S)|$ is strict in this particular case. In fact, we can prove that the bound is strict for every $m$ and not just for $m = 2^k$. Consider the following Procedure 1 that generates a set $S$ of $T(m)$ strings of size $m$.

**Procedure 1** $f[m]$

```
begin
    if $m = 1$
    then return(1);
    else begin
        $h_1 = \lceil m/2 \rceil$; $h_0 = \lfloor m/2 \rfloor$;
        $S_0 = f[h_0]$; $S_1 = S_0$;
        if $h_1 \neq h_0$ then $S_1 := (\texttt{0}, \texttt{0}, \ldots, \texttt{0}) \cup S_1$;
        $S_0 := \texttt{0} \circ S_0$;
        $S_1 := \texttt{1} \circ S_1$;
        return($S_0 \cup S_1$);
    end
end
```

These are the sets produced by the procedure for $m = 1, \ldots, 6$, for which

$$m = 1 \to S = \{\texttt{1}\} \to P(S) = S = \{\texttt{1}\}$$
$$m = 2 \to S = \{\texttt{01}, \texttt{11}\} \to P(S) = S \cup \{\texttt{-1}\} = \{\texttt{01}, \texttt{11}, \texttt{-1}\}$$
$$m = 3 \to S = (\texttt{01}, \texttt{10}, \texttt{11}) \to P(S) = S \cup (\texttt{-1}, \texttt{1-})$$
$$m = 4 \to S = (\texttt{001}, \texttt{011}, \texttt{101}, \texttt{111}) \to P(S) = S \cup (\texttt{0-1}, \texttt{-01}, \texttt{1-1}, \texttt{-11}, \texttt{--1})$$
$$m = 5 \to S = (\texttt{001}, \texttt{011}, \texttt{100}, \texttt{101}, \texttt{111}) \to P(S) = S \cup (\texttt{0-1}, \texttt{10-}, \texttt{1-1}, \texttt{-01}, \texttt{-11}, \texttt{--1})$$
$$m = 6 \to S = (\texttt{001}, \texttt{010}, \texttt{011}, \texttt{101}, \texttt{110}, \texttt{111}) \to$$
$$P(S) = S \cup (\texttt{01-}, \texttt{0-1}, \texttt{11-}, \texttt{1-1}, \texttt{-01}, \texttt{-10}, \texttt{-11}, \texttt{-1-}, \texttt{--1})$$

It is easy to show that $|P(S)| = T(m)$ for the sets produced by the above procedure, and so the bound $T(m)$ is strict for all $m$. However, in most cases, as for covering problems, we may be interested only in maximal patterns $P^*(S)$ and we may wonder how the number of maximal patterns grows with $m$.

In the particular case of the strings produced by the above procedure, which corresponds to the worst case in terms of generic patterns, the number of maximal patterns grows very slowly, indeed as $\log m$.

In the following table we report the value $m$, $|P(S)| = T(m)$, and the number $|P^*(S)|$ of maximal patterns for the above set of strings

$$\left(\begin{array}{cccccccccccccccccc} m & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ |P(S)| & 1 & 3 & 5 & 9 & 11 & 15 & 19 & 27 & 29 & 33 & 37 & 45 & 49 & 57 & 65 & 89 \\ |P^*(S)| & 1 & 1 & 2 & 1 & 2 & 2 & 3 & 1 & 2 & 2 & 3 & 2 & 3 & 3 & 4 & 1 \end{array}\right)$$

It turns out that $|P^*(S)|$ is equal to the number of ones in the binary expression of $m$. However, the number $|P^*(S)|$ can be much larger for other sets of strings. There are cases such that $|P^*(S)| > m$. For instance, consider the following case for which $|S| = 12$, $|P(S)| = 25 < T(12) = 45$ and $|P^*(S)| = 13$:

$$S = (\mathtt{00000, 00001, 00101, 01000, 01010, 10000, 10010, 10101, 10110, 10111, 11010, 11101})$$

$$P(S) = \begin{pmatrix} \mathtt{00000} & \mathtt{00001} & \mathtt{0000-} & \mathtt{00101} & \mathtt{00-01} & \mathtt{01000} & \mathtt{01010} & \mathtt{010-0} & \mathtt{0-000} \\ \mathtt{10000} & \mathtt{10010} & \mathtt{100-0} & \mathtt{10101} & \mathtt{10110} & \mathtt{10111} & \mathtt{1011-} & \mathtt{101-1} & \\ \mathtt{10-10} & \mathtt{11010} & \mathtt{11101} & \mathtt{1-010} & \mathtt{1-101} & \mathtt{-0000} & \mathtt{-0101} & \mathtt{-1010} & \end{pmatrix}$$

$$P^*(S) = \begin{pmatrix} \mathtt{0000-} & \mathtt{00-01} & \mathtt{010-0} & \mathtt{0-000} & \mathtt{100-0} & \mathtt{1011-} & \mathtt{101-1} \\ \mathtt{10-10} & \mathtt{1-010} & \mathtt{1-101} & \mathtt{-0000} & \mathtt{-0101} & \mathtt{-1010} & \end{pmatrix}$$

Although it may happen that $|P^*(S)| > m$ in general, we may show that a cover can always be obtained by less than $m$ patterns if for each string there is a compatible pattern with at least one gap that covers it. Let us call a one-pattern a pattern with exactly one gap. Consider the set of all compatible one-patterns for a given set of $m$ strings of $n$ symbols. By assumption, each string is covered by a one-pattern (if it is covered by a pattern with gaps it is also covered by a one-pattern). Build a graph $G = (V, E)$ with $V$ the set of strings and $E$ the set of string pairs spanned by a one-pattern. This graph is bipartite because we may partition the strings into "even" and "odd" strings according to the number of ones in the string. A one-pattern necessarily spans an even string and an odd one. Moreover, there are no isolated vertices by assumption. A cover consisting of one-patterns corresponds to an edge cover of $G$. The cardinality of an edge cover is given by $m - |M|$ with $M$ the cardinality of a maximum matching. Since $|M| \geq 1$ we need at most $m - 1$ one-patterns to cover all strings. For the above example these are two alternative minimum covers with $6 < 12$ patterns.

$$(\mathtt{00-01, 010-0, 1011-, 1-010, 1-101, -0000}), \quad (\mathtt{00-01, 0-000, 100-0, 1011-, 1-101, -1010})$$

Therefore, if more than $m$ patterns are needed to cover a set of $m$ strings, this means that there are some special strings that, loosely speaking, cannot be explained by some rule and require a particular pattern that coincides with the string. Furthermore, the $m - 1$ bound is obtained by considering only one-patterns. If, as we presume, the data sets are explained by more interesting patterns with many gaps, the size of a minimum cover can be significantly less than $m$.

## 5. ILP Models

In this section we provide ILP models for two of the problems defined in Section 3, namely PATTERN COVER MINIMALITY and PATTERN EQUIVALENCE. The first problem is NP-complete and the second one is co-NP-complete. So it is not inappropriate to use ILP models for their solution. On the contrary, we believe that PATTERN EQUIVALENCE MINIMALITY cannot be expressed as an ILP model. We have already stressed the fact that this problem is both NP-hard and co-NP-hard and we have also observed that, given the current state of the art in computational complexity, we believe that it does not belong to NP or co-NP. Since ILP problems belong to these classes, we have strong reasons to doubt about the possibility of solving PATTERN EQUIVALENCE MINIMALITY by ILP models.

### *5.1. ILP for Pattern Cover Minimality*

We approach the problem of finding a pattern set $P$ of minimum cardinality that spans a given set of strings $S$ as a 01LP set cover problem, in which each row is associated to each string of the string set, each column is associated to each compatible pattern for $S$, and the entry $a_{ij}$ of the 01 LP matrix is 1 if and only if the pattern $j$ covers the string $i$. In view of Theorem 2 the matrix has a polynomial number of columns and therefore it can be explicitly written. We note that it is not strictly necessary to generate the full matrix. We may use a column generation approach by adapting the algorithm that generates all pattern to the pricing problem given dual variables associated to the strings. However, in our computational experiments we have seen that generating the full matrix and then solving the problem outperforms the column generation approach, which requires running the recursive algorithm for each column generation, while only one run is necessary for generating the full matrix.

### *5.2. ILP for Pattern Equivalence*

We assume that two sets $P$ and $Q$ of patterns are given. We introduce the following models

$$\begin{aligned} v = \min \quad & \sum_{q \in Q} z_q \\ & \sum_{i:q_i=0} x_i + \sum_{i:q_i=1} (1 - x_i) \geq 1 - z_q & & q \in Q \\ & y_p \leq 1 - x_i & & i : p_i = 0, \quad p \in P \\ & y_p \leq x_i & & i : p_i = 1, \quad p \in P \\ & \sum_{p \in P} y_p \geq 1 \\ & x_i \in \{0,1\}, y_p \in \{0,1\}, z_q \geq 0 \text{ integer} \end{aligned} \tag{3}$$

$$\begin{aligned} w = \min \quad & \sum_{p \in P} y_p \\ & \sum_{i:p_i=0} x_i + \sum_{i:p_i=1} (1 - x_i) \geq 1 - y_p & & p \in P \\ & z_q \leq 1 - x_i & & i : q_i = 0, \quad q \in Q \\ & z_q \leq x_i & & i : q_i = 1, \quad q \in Q \\ & \sum_{q \in Q} z_q \geq 1 \\ & x_i \in \{0,1\}, z_q \in \{0,1\}, y_p \geq 0 \text{ integer} \end{aligned} \tag{4}$$

We have the following result:

**Proposition 7.**
*– $S(P) \subset S(Q)$ if and only if $v > 0$ and $w = 0$;*
*– $S(Q) \subset S(P)$ if and only if $w > 0$ and $v = 0$;*
*– $S(Q) = S(P)$ if and only if $v > 0$ and $w > 0$.*

**Proof.** We reiterate that $\subset$ means strict inclusion. It is sufficient to prove that $S(P) \subseteq S(Q)$ if and only if $v > 0$. If $y_p = 1$ then $x$ is generated by $p \in P$. The constraint $\sum_{p \in P} y_p \geq 1$ implies that $x$ is generated by at least one pattern in $P$. Hence feasible $x$ are in $S(P)$. Consider now any $x \in \{0,1\}^n$. If $x$ is generated by $q \in Q$ then $z_q = 1$, while if $x$ is not generated by $q \in Q$ then $z_q = 0$ is feasible (along with possible integer values $z_q \geq 1$). The objective function forces $z_q$ to be zero in this case.

Therefore, $v = 0$ if and only if $x \in S(P)$ and $x \notin S(Q)$. If $v > 0$, for any pattern $x \in S(P)$ we have that $x \in S(Q)$, i.e., $S(P) \subseteq S(Q)$. $\square$

Note that, if $S(P) \not\subseteq S(Q)$, i.e., when $v = 0$, the model (3) yields also a string $x$ in $S(P)$ but not in $S(Q)$, whereas if $S(P) \subseteq S(Q)$, i.e., when $v > 0$, model (3) yields also a string $x$ in both $S(P)$ and $S(Q)$. Similarly if $S(Q) \not\subseteq S(P)$, i.e., when $w = 0$, model (4) yields also a string $x$ in $S(Q)$ but not in $S(P)$, whereas if $S(Q) \subseteq S(P)$, i.e., when $w > 0$, model (4) yields also a string $x$ in both $S(Q)$ and $S(P)$.

We may further distinguish the case $w = 0$, $v = 0$, via the following model

$$\begin{aligned}
\hat{w} = \min \quad & w_p + w_q \\
& y_p \le 1 - x_i && i : p_i = 0, \quad p \in P \\
& y_p \le x_i && i : p_i = 1, \quad p \in P \\
& \sum_{p \in P} y_p \ge 1 - w_p \\
& z_q \le 1 - x_i && i : q_i = 0, \quad q \in Q \\
& z_q \le x_i && i : q_i = 1, \quad q \in Q \\
& \sum_{q \in Q} z_q \ge 1 - w_q
\end{aligned} \tag{5}$$

The following proposition follows easily from Proposition 7.

**Proposition 8.** *$S(P)$ and $S(Q)$ are disjoint if and only if $\hat{w} > 0$.*

When $S(P)$ and $S(Q)$ are not disjoint, the model yields a string $x$ shared by both sets.

If we consider model (4) and take $Q = \{(\texttt{-}\,\texttt{-}\cdots\texttt{-})\}$ we are actually solving the problem FULL PATTERN COVERAGE together with its complement PARTIAL PATTERN COVERAGE. Hence (4) becomes

$$\begin{aligned}
u = \min \quad & \sum_{p \in P} y_p \\
& \sum_{i:p_i=0} x_i + \sum_{i:p_i=1} (1 - x_i) \ge 1 - y_p && p \in P \\
& x_i \in \{0,1\}, y_p \ge 0 \text{ integer}
\end{aligned} \tag{6}$$

and we may conclude, as a Corollary of Proposition 7, that

**Proposition 9.** *$S(P) = \{0, 1\}^n$ if and only if $u > 0$.*

As a simple example of the previous results suppose we are given the two following sets of patterns $P$ and $Q$.

$$P = \begin{bmatrix} \texttt{- - 1 1} \\ \texttt{1 0 0 -} \\ \texttt{0 0 - -} \\ \texttt{- 1 - 1} \end{bmatrix} \qquad Q = \begin{bmatrix} \texttt{0 0 - 0} \\ \texttt{0 - 1 1} \\ \texttt{- 0 - 1} \end{bmatrix}$$

We run in sequence (3) and (4) and obtain $v = 0$ and $w > 0$, that implies $S(Q) \subset S(P)$, according to Proposition 7. In this case there is no need of running (5). As a byproduct we obtain from (3) and (4) respectively the strings

$$x^1 = (\texttt{1111}) \qquad\qquad x^2 = (\texttt{0010})$$

One can easily check that $x^1 \in S(P)$ and $x^1 \notin S(Q)$ and also that $x^2 \in S(Q) \subset S(P)$. Moreover, if we run (6) for $P$ we obtain $u > 0$, that implies $S(P) \neq S((\texttt{----}))$ and also the string $(\texttt{1110})$ as a certificate that $S(P)$ does not contain all strings.

If we are given the two sets of patterns

$$P = \begin{bmatrix} \texttt{- 1 1 1} \\ \texttt{1 0 0 -} \\ \texttt{0 0 0 -} \\ \texttt{- 1 - 1} \end{bmatrix} \qquad Q = \begin{bmatrix} \texttt{0 0 - 0} \\ \texttt{0 - 1 1} \\ \texttt{- 0 - 1} \end{bmatrix}$$

and run in sequence (3) and (4) we obtain $v = 0$ and $w = 0$. Hence we have to run also (5) and obtain $\hat{w} = 0$. This means that $S(P)$ and $S(Q)$ are not disjoint. We may exhibit also the string $x^3 \in S(P) \cap S(Q)$ that belongs to their intersection. Moreover, from the previous (3) and (4) we also have the two strings $x^1 \in S(P)$, $x^1 \notin S(Q)$ and $x^2 \in S(Q)$, $x^2 \notin S(P)$. As a final output we may run (6) for $P \cup Q$ and obtain $u = 0$ and $x^4 \notin S(P) \cup S(Q)$ :

$$x^1 = (\mathtt{1111}) \qquad x^2 = (\mathtt{0010}) \qquad x^3 = (\mathtt{0001}) \qquad x^4 = (\mathtt{1110})$$

## 6. Computational Experiments

We have carried out computational experiments for PATTERN COVER MINIMALITY and PATTERN EQUIVALENCE. The problem PATTERN COVER is polynomial and we felt no need to perform computational experiments for this problem. On the opposite side, problem PATTERN EQUIVALENCE MINIMALITY seems to be intractable and we have not even devised ideas of how to solve it. Problems PATTERN COMPLETENESS and PATTERN INCOMPLETENESS are particular cases of PATTERN EQUIVALENCE.

Our tests were run on an Intel Core i5 machine 2.3 GHz with 8 GB Ram. The program was implemented in C++ and we used Cplex 12.4 as the ILP solver.

### *6.1. Pattern Cover Minimality*

We approach the problem of finding a pattern set $P$ of minimum cardinality that spans a given set $S$ of strings as a 01LP set cover problem, in which each row is associated to each string of the string set, each column is associated to each compatible pattern for $S$, and the entry $a_{ij}$ of the 01 LP matrix is 1 if and only if the pattern $j$ covers the string $i$. In view of Theorem 2 the matrix has a polynomial number of columns and therefore it can be explicitly written. We note that it is not strictly necessary to generate the full matrix. We may use a column generation approach by adapting the algorithm that generates all patterns to the pricing problem given dual variables associated to the strings. However, we have seen that generating the full matrix and then solving the problem outperforms the column generation approach, which requires running the recursive algorithm for each column generation, while only one run is necessary for generating the full matrix.

We fix the size of a string to $n = 15$. Each string is randomly generated by independently setting each bit to 1 with probability $p$ (and to 0 with probability $1 - p$). A random instance consists of a set $S$ of $m$ randomly generated strings without duplicate strings. We consider the following values: $p \in \{0.1, 0.25, 0.5\}$ and $m \in \{100, 1000, 5000, 10,000\}$. For each combination of values of $p$ and $m$ we generate ten instances.

The strings generated with a value of $p$ close to 0 (or equivalently close to 1) tend to be similar whereas they are much less similar for $p = 0.5$. Similar instances are expected to be covered with a few patterns with many gaps, whereas non-similar instances are expected to be covered with many patterns with few gaps.

By the recursive procedure described in Section 4 we compute for each $S$ all compatible patterns $P(S)$ and solve with cplex the corresponding set cover problem.

The computational results are reported in Table 1. For each combination of $p$ and $m$ we report for each one of the ten instances the resulting number of compatible patterns ($|P(S)|$), the optimal value of the minimal cover problem (opt), the total cpu time in seconds consisting of the pattern generation procedure plus the cplex run (time), and the number of nodes (root excluded) of the branch-and-bound process (#nodes). A value of #nodes equal to zero means that the solution of the LP relaxation was already integer.

**Table 1.** Results for Pattern Cover Minimality.

| | $p = 0.1$ | | | | $p = 0.25$ | | | | $p = 0.5$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m$ | $\|P(S)\|$ | **Opt** | **Time** | **#nod** | $\|P(S)\|$ | **Opt** | **Time** | **#nod** | **#pat** | **Opt** | **Time** | **#nod** |
| | 300 | 60 | 0.038 | 0 | 136 | 83 | 0.002 | 0 | 102 | 98 | 0.001 | 0 |
| | 341 | 57 | 0.023 | 0 | 137 | 84 | 0.002 | 0 | 104 | 96 | 0.001 | 0 |
| | 322 | 58 | 0.009 | 0 | 119 | 88 | 0.002 | 0 | 103 | 97 | 0.001 | 0 |
| | 314 | 56 | 0.011 | 0 | 117 | 91 | 0.003 | 0 | 103 | 97 | 0.004 | 0 |
| | 302 | 66 | 0.008 | 0 | 123 | 85 | 0.002 | 0 | 104 | 96 | 0.002 | 0 |
| 100 | 298 | 58 | 0.007 | 0 | 139 | 80 | 0.003 | 0 | 102 | 98 | 0.002 | 0 |
| | 317 | 55 | 0.014 | 0 | 112 | 92 | 0.002 | 0 | 102 | 98 | 0.001 | 0 |
| | 298 | 58 | 0.008 | 0 | 126 | 82 | 0.008 | 0 | 103 | 97 | 0.002 | 0 |
| | 292 | 63 | 0.008 | 0 | 128 | 85 | 0.002 | 0 | 101 | 99 | 0.002 | 0 |
| | 282 | 66 | 0.011 | 0 | 129 | 87 | 0.002 | 0 | 100 | 100 | 0.002 | 0 |
| | 9118 | 380 | 0.220 | 0 | 3729 | 546 | 0.103 | 0 | 1234 | 843 | 0.031 | 0 |
| | 9131 | 381 | 0.219 | 0 | 3770 | 544 | 0.097 | 0 | 1197 | 853 | 0.031 | 0 |
| | 8567 | 400 | 0.215 | 0 | 3912 | 550 | 0.092 | 0 | 1245 | 825 | 0.032 | 0 |
| | 8963 | 384 | 0.205 | 0 | 3704 | 557 | 0.093 | 0 | 1213 | 845 | 0.034 | 0 |
| | 8757 | 392 | 0.215 | 0 | 3540 | 565 | 0.085 | 0 | 1214 | 838 | 0.035 | 0 |
| 1000 | 8651 | 396 | 0.203 | 0 | 3722 | 543 | 0.098 | 0 | 1218 | 837 | 0.047 | 0 |
| | 8735 | 369 | 0.207 | 0 | 3813 | 553 | 0.095 | 0 | 1226 | 838 | 0.031 | 0 |
| | 8888 | 389 | 0.225 | 0 | 3520 | 556 | 0.094 | 0 | 1219 | 846 | 0.030 | 0 |
| | 8709 | 389 | 0.220 | 0 | 3636 | 572 | 0.101 | 0 | 1232 | 838 | 0.034 | 0 |
| | 8874 | 381 | 0.349 | 0 | 3677 | 566 | 0.098 | 0 | 1224 | 828 | 0.056 | 0 |
| | 117,558 | 1323 | 7.282 | 0 | 70,726 | 1614 | 5.421 | 0 | 11,211 | 2767 | 0.758 | 0 |
| | 118,365 | 1277 | 7.080 | 0 | 69,997 | 1614 | 4.771 | 0 | 11,206 | 2752 | 0.801 | 0 |
| | 116,936 | 1282 | 6.668 | 0 | 68,989 | 1623 | 3.882 | 0 | 11,232 | 2770 | 0.746 | 0 |
| | 118,087 | 1281 | 6.592 | 0 | 69,964 | 1608 | 9.634 | 0 | 11,144 | 2757 | 0.708 | 0 |
| | 118,435 | 1282 | 9.959 | 0 | 68,920 | 1618 | 4.576 | 0 | 11,042 | 2787 | 0.737 | 0 |
| 5000 | 115,737 | 1293 | 6.254 | 0 | 68,609 | 1631 | 5.847 | 0 | 11,284 | 2755 | 0.746 | 0 |
| | 117,605 | 1301 | 6.473 | 0 | 68,655 | 1626 | 5.419 | 0 | 11,226 | 2753 | 0.711 | 0 |
| | 116,015 | 1301 | 6.539 | 0 | 68,483 | 1631 | 4.817 | 0 | 11,136 | 2786 | 0.685 | 0 |
| | 116,910 | 1252 | 6.996 | 0 | 68,242 | 1598 | 9.475 | 0 | 11,213 | 2744 | 0.708 | 0 |
| | 116,534 | 1301 | 7.444 | 0 | 69,466 | 1613 | 7.985 | 0 | 11,317 | 2749 | 0.747 | 0 |
| | 431876 | 1944 | 52.0 | 0 | 295,360 | 2197 | 397.0 | 0 | 40,574 | 3740 | 281.4 | 1302 |
| | 430,941 | 1989 | 47.8 | 0 | 295,703 | 2250 | 435.9 | 0 | 40,647 | 3728 | 513.8 | 1020 |
| | 433,162 | 1997 | 39.3 | 0 | 294,783 | 2236 | 388.0 | 0 | 40,664 | 3738 | 485.2 | 847 |
| | 426,007 | 1992 | 60.8 | 0 | 304,432 | 2199 | 397.3 | 0 | 40,438 | 3759 | 333.1 | 914 |
| | 431,029 | 1996 | 66.7 | 0 | 293,031 | 2255 | 404.6 | 0 | 40,467 | 3728 | 1004.4 | 2254 |
| 10,000 | 434,909 | 1935 | 40.8 | 0 | 295,167 | 2238 | 442.9 | 0 | 40,686 | 3751 | 1141.8 | 6319 |
| | 431,853 | 1943 | 40.6 | 0 | 293,073 | 2221 | 418.7 | 0 | 40,134 | 3765 | 307.8 | 729 |
| | 430,508 | 1951 | 70.7 | 0 | 297,234 | 2242 | 386.9 | 0 | 40,253 | 3750 | 381.1 | 1163 |
| | 431,020 | 1958 | 58.7 | 0 | 292,270 | 2230 | 399.4 | 0 | 40,793 | 3696 | 2608.6 | 4682 |
| | 432,664 | 1957 | 44.1 | 0 | 293,697 | 2233 | 442.3 | 0 | 40,538 | 3703 | 1802.4 | 3784 |

As can be seen from Table 1, all instances are solved at the branch-and-bound root node except the case $p = 0.5$ and $m = 10{,}000$.

### *6.2. Pattern Equivalence*

One of the main difficulties for testing the models for PATTERN EQUIVALENCE is creating sensible instances which show that the ILP model is indeed effective.

In fact, a major objection that one might have versus the use of ILP is that—when the maximum number of gaps in the patterns is not "large enough"—a simple enumerative approach might prove quite effective, and much better than ILP, even if there are a lot of patterns and $n$ is quite big. Assume, for example, to compare two sets of patterns of about 1000 patterns each with $n = 100$, and each pattern has at most ten "-" in it. Ten gaps can be expanded in 1024 ways, and so each set of patterns yields at most about 1,000,000 strings, which most computers can generate in a second. Then, we just need to check whether these two sets of strings have the same size (if not, we stop) and, if they do, we compare each element of the first to each one of the second and stop as soon as one element of the first is not in the second (perhaps by first sorting the two sets and then scanning the sorted lists). Some data structures might be more effective than others for these operations, but, bottom line, it is a very fast process that ILP has a hard time beating.

Therefore, we want to show that ILP is the way to go when the naïve approach cannot work, namely, when the patterns have so many gaps in them that a complete expansion (which exponentially increases the data size) is out of question. This poses the problem of how to create non-trivial, interesting instances of equivalent pattern sets which have a large number of gaps.

## 6.3. Diagonal Instances

A simple way of creating instances with equivalent pattern sets is as follows. For every $n$, we consider two equivalent sets of patterns, which generate all strings with the exception of the string $\texttt{11}\cdots\texttt{11}$. We call these *diagonal instances*.

The first set has $n$ patterns, with a maximum number of gaps $n-1$, and is the following (exemplified for $n = 6$):

$$\begin{bmatrix}\texttt{0-----}\\\texttt{-0----}\\\texttt{--0---}\\\texttt{---0--}\\\texttt{----0-}\\\texttt{-----0}\end{bmatrix}$$

The second set has $2n-1$ patterns, with a maximum number of gaps $n-2$, and is

$$\begin{bmatrix}\texttt{01----}\\\texttt{0-1---}\\\texttt{0--1--}\\\texttt{0---1-}\\\texttt{0----1}\\\texttt{10----}\\\texttt{1-0---}\\\texttt{1--0--}\\\texttt{1---0-}\\\texttt{1----0}\\\texttt{000000}\end{bmatrix}$$

We perform a sequence of tests to compare the ILP approach with the complete enumeration algorithm, for increasing values of $n$. These diagonal instances turn out to be very easy for the ILP model. They are all solved in less than 0.1 s, for $n \leq 30$ as can be seen from Table 2. The enumerative approach, however, becomes very soon impractical. For $n = 28$ the algorithm takes already more than 15 min, while for $n = 30$ the algorithm has not finished after one hour (which we set as a maximum time limit). It is interesting to notice how the ILP approach solves this instance in less than a second also for $n = 100$, while the enumerative approach would have to generate $2^{100} - 1$ strings.

**Table 2.** Computational results for Pattern Equivalence—Diagonal instances.

| $n$ | ILP | enum |
|---|---|---|
| 18 | 0.03 s | 0.34 s |
| 20 | 0.05 s | 1.38 s |
| 22 | 0.07 s | 6.43 s |
| 24 | 0.08 s | 29.89 s |
| 26 | 0.08 s | 138 s |
| 28 | 0.05 s | 1005 s |
| 30 | 0.08 s | did not finish |

### *6.4. Generating Equivalent Pattern Sets in General*

We can adopt the following strategy to create two sets of equivalent patterns:

1. We generate a small starting set $P$ (e.g., $|P| = 3, 4$) of random patterns. Each pattern is obtained by setting each bit to "-" with some probability $q$, to 0 with some probability $p < 1 - q$, and to 1 with probability $1 - p - q$. Since we are interested in patterns with many gaps, we set $q$ to a large value (e.g., 0.8). Let $g$ be the minimum number of gaps appearing in some pattern of $P$.
2. The patterns in $P$ are expanded in all possible ways yielding a set $S$ of strings.
3. We compute with the recursive procedure described in Section 4 (slightly modified) the set $\mathcal{P}$ of all patterns compatible with $S$ which have at least $g$ gaps each (this ensures that it is always possible to cover $S$ with patterns in $\mathcal{P}$).
4. We compute two random solutions of the set covering problem. Namely, from $\mathcal{P}$ we select (picking patterns at random until we have a cover) two subsets $P_1, P_2$ that are covers of $S$.
5. $P_1, P_2$ are equivalent by construction and have a fairly large number of gaps in each pattern.

In our implementation, because of memory problems, the above procedure works for $n \leq 20$. Thus, to build larger instances we use a trick. Namely, we create instances starting from instances built as above and then combining them into larger and larger ones as explained below.

### *6.5. How to Boost Instances of Pattern Equivalence*

One way to increase the number of gaps in the instances would be to take two set of equivalent patterns $A$ and $B$ and suffix each pattern with a list of $k$ gaps. This, however, yields very particular, uninteresting, instances. In order to obtain more elaborate, hard pattern equivalence instances, we have developed the following scheme.

Given a set of patterns $X$, denote by

- $n(X)$ the number of columns (i.e., string length),
- $m(X)$ be the number of rows (i.e., of patterns),
- $G(X)$ the maximum number of gaps in some pattern.

Furthermore, given sets of patterns $A$ and $B$, denote by $C = A \times B$ the set of patterns

$$C = \{(a, b) : a \in A, b \in B\}$$

Note that $n(C) = n(A) + n(B)$, $m(C) = m(A) \cdot m(B)$ and $G(C) = G(A) + G(B)$. We have

**Claim 1.** *Let $A_1, A_2, B_1, B_2$ be sets of patterns such that $A_1$ is equivalent to $A_2$ and $B_1$ is equivalent to $B_2$. Then $A_1 \times B_1$ is equivalent to $A_2 \times B_2$.*

**Proof.** Let $C_i := A_i \times B_i$. We want to show that $S(C_1) = S(C_2)$. We show $S(C_1) \subset S(C_2)$ since the other direction is symmetrical. Let $(x, y) \in S(C_1)$. In particular, $x \in S(A_1)$ and

$y \in S(B_1)$. Since $A_1$ is equivalent to $A_2$ also $x \in S(A_2)$ and similarly $y \in S(B_2)$. Hence $(x, y) \in S(A_1) \times S(B_2) = S(C_2)$. $\square$

By using this trick repeatedly, we can snowball from small instances, e.g., 4 or 5 patterns with 5 or 6 gaps each, to instances with a few hundred patterns with more than 20 gaps each.

### *6.6. Experiments*

We have created 10 instances of size $n = 30$ each. Each instance is built by combining two equivalent instances of size $n = 15$ each, which were built with our procedure with parameters $q = 0.8$, $p = 0.1$ and $|P| \in \{3, 4\}$. The results are reported in Table 3. These instances turned out to be too difficult to be solved by the enumerative approach in less than half hour each.

**Table 3.** Computational results for Pattern Equivalence—boosted instances.

| $m_1$ | $g_1$ | $G_1$ | $a_1$ | $m_2$ | $g_2$ | $G_2$ | $a_2$ | **Time(s)** | **BB Nodes1** | **BB Nodes 2** |
|---|---|---|---|---|---|---|---|---|---|---|
| 264 | 22 | 23 | 22.18 | 266 | 22 | 24 | 22.19 | 0.43 | 0 | 23 |
| 270 | 24 | 25 | 24.03 | 150 | 24 | 25 | 24.04 | 0.55 | 10 | 21 |
| 273 | 22 | 22 | 22.00 | 220 | 22 | 23 | 22.13 | 0.70 | 0 | 327 |
| 266 | 23 | 24 | 23.10 | 279 | 23 | 26 | 23.20 | 2.21 | 48 | 275 |
| 735 | 22 | 24 | 22.15 | 1089 | 22 | 25 | 22.18 | 4.02 | 30 | 264 |
| 342 | 23 | 25 | 23.32 | 667 | 23 | 25 | 23.07 | 4.28 | 684 | 0 |
| 456 | 22 | 24 | 22.30 | 453 | 22 | 25 | 22.29 | 15.16 | 2125 | 1512 |
| 735 | 22 | 25 | 22.35 | 540 | 22 | 25 | 22.23 | 50.52 | 927 | 3095 |
| 688 | 20 | 21 | 20.09 | 360 | 20 | 21 | 20.16 | 76.24 | 1694 | 1940 |
| 784 | 21 | 22 | 21.18 | 261 | 21 | 24 | 21.64 | 172.28 | 13504 | 1811 |

For each set of input patterns ($i = 1, 2$) we have:

1. $m_i$ is the number of input patterns.
2. $g_i$ is the minimum number of gaps per pattern.
3. $G_i$ is the maximum number of gaps per pattern.
4. $a_i$ is the average number of gaps per pattern.

The ten instances are reported in Table 3 sorted by running times (9-th column). The 10-th and 11-th columns report the number of branch-and-bound nodes required (root node excluded) for solving models (3) and (4), respectively.

The results show that the ILP approach is effective also for instances which are large and enough and cannot be tackled by enumerative approaches.

In order to test the same ILP models in case the pattern sets are not equivalent we have randomly perturbed the previous data. The running times remained practically the same.

## 7. Conclusions

In order to use feature selection and LAD in the analysis of binary data consisting of positive and negative samples, one has to identify which computational problems might arise and how to overcome them. One of the issues that we have addressed in this paper is in fact the computational complexity of the problems, which we have shown to be, in general, very hard. As a viable approach to the effective solution of some of these problems, we have described integer linear programming formulations. In particular, we have given ILP models for the problem of determining if two sets of patterns are equivalent and for finding a min-size set of patterns which explain a given data set. A striking consequence of our complexity results is that there could be no simple ILP model for finding a minimal set of patterns explaining the same data set explained by a given pattern set. Developing some procedures for this last problem could be a line of future research.

**Author Contributions:** Conceptualization, methodology, software, validation, writing—original draft, review and editing: G.L. and P.S. Both authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Jaiwei, H.; Jian, P.; Micheline, K. *Data Mining: Concepts and Techniques*; Morgan Kaufmann: Burlington, MA, USA, 2011.
2. Kantardzic, M. *Data Mining: Concepts, Models, Methods, and Algorithms*; John Wiley & Sons: Hoboken, NJ, USA, 2003.
3. Dash, M.; Liu, H. Feature Selection for Classification. *Intell. Data Anal.* **1997**, *1*, 131–156. [CrossRef]
4. Felici, G.; de Angelis, V.; Mancinelli, G. Feature Selection for Data Mining. In *Data Mining and Knowledge Discovery Approaches Based on Rule Induction Techniques*; Felici, G., Triantaphyllou, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 227–252.
5. Stanczyk, U.; Zielosko, B.; Jain, L.C. *Advances in Feature Selection for Data and Pattern Recognition: An Introduction Advances in Feature Selection for Data and Pattern Recognition*; Intelligent Systems Reference Library; Stacczyk, U., Zielosko, B., Jain, L., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 138.
6. Alexe, G.; Alexe, S.; Bonates, T.O.; Kogan, A. Logical analysis of data—The vision of Peter L. Hammer. *Ann. Math. Artif. Intell.* **2007**, *49*, 265–312. [CrossRef]
7. Chikalov, I.; Lozin, V.; Lozina, I.; Moshkov, M.; Son Nguyen, H.; Skowron, A.; Zielosko, B. Logical Analysis of Data: Theory, Methodology and Applications. In *Three Approaches to Data Analysis*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 147–192
8. Hammer, P.; Bonates, T. *Logical Analysis of Data: From Combinatorial Optimization to Medical Applications*; RUTCOR Research Report, 10-05; Rutgers University: New Brunswick, NJ, USA, 2005.
9. Bertolazzi, P.; Felici, G.; Festa, P.; Lancia, G. Logic classification and feature selection for biomedical data. *Comput. Math. Appl.* **2008**, *55*, 889–899. [CrossRef]
10. Golub, T.R.; Slonim, D.K.; Tamayo, P.; Huard, C.; Gaasenbeek, M.; Mesirov, J.P.; Coller, H.; Loh, M.L.; Downing, J.R.; Caligiuri, M.A.; et al. Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. *Science* **1999**, *286*, 531–537. [CrossRef] [PubMed]
11. Li, T.; Zhang, C; Ogihara, M. A comparative study of feature selection and multiclass classification methods for tissue classification based on gene expression. *Bioinformatics* **2004**, *20*, 2429–2437. [CrossRef]
12. Lancia, G.; Serafini, P. The Complexity of Some Pattern Problems in the Logical Analysis of Large Genomic Data Sets. In *Bioinformatics and Biomedical Engineering. IWBBIO 2016*; Lecture Notes in Computer Science; Ortuno, F., Rojas, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9656.
13. Lancia, G.; Mathieson, L.; Moscato, P. Separating sets of strings by finding matching patterns is almost always hard. *Theor. Comput. Sci.* **2017**, *665*, 73–86. [CrossRef]
14. Boccia, M.; Sforza, A.; Sterle, C. Simple Pattern Minimality Problems: Integer Linear Programming Formulations and Covering-Based Heuristic Solving Approaches. *Informs J. Comput.* **2020**. [CrossRef]
15. Serafini, P. Classifying negative and positive points by optimal box clustering. *Discret. Appl. Math.* **2014**, *165*, 270–282. [CrossRef]
16. Boros, E.; Hammer, P.; Ibaraki, T.; Kogan, A.; Mayoraz, E.; Muchnik, I. An implementation of Logical Analysis of Data. *IEEE Trans. Knowl. Data Eng.* **2000**, *12*, 292–306. [CrossRef]
17. Garey, M.R.; Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; W.H. Freeman and Company: San Francisco, CA, USA, 1979.
18. Cormen, T.; Leiserson, C.E.; Rivest,R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009.