Proceedings of Machine Learning Research vol 65:1–23, 2017

Learning Disjunctions of Predicates

Nader H. Bshouty BSHOUTY@CS.TECHNION.AC.IL
Technion

Dana Drachsler-Cohen DDANA@CS.TECHNION.AC.IL
Technion

Martin Vechev MARTIN.VECHEV@INF.ETHZ.CH
ETH Zurich

Eran Yahav YAHAVE@CS.TECHNION.AC.IL
Technion

Abstract

Let $\mathcal{F}$ be a set of boolean functions. We present an algorithm for learning $\mathcal{F}_\vee := \{\vee_{f\in S} f \mid S \subseteq \mathcal{F}\}$ from membership queries. Our algorithm asks at most $|\mathcal{F}| \cdot \mathrm{OPT}(\mathcal{F}_\vee)$ membership queries where $\mathrm{OPT}(\mathcal{F}_\vee)$ is the minimum worst case number of membership queries for learning $\mathcal{F}_\vee$. When $\mathcal{F}$ is a set of halfspaces over a constant dimension space or a set of variable inequalities, our algorithm runs in polynomial time.

The problem we address has practical importance in the field of program synthesis, where the goal is to synthesize a program that meets some requirements. Program synthesis has become popular especially in settings aiming to help end users. In such settings, the requirements are not provided upfront and the synthesizer can only learn them by posing membership queries to the end user. Our work enables such synthesizers to learn the exact requirements while bounding the number of membership queries.

1. Introduction

Learning from membership queries (Angluin, 1988) has flourished due to its many applications in group testing (Du and Hwang, 2000, 2006), blood testing (Dorfman, 1943), chemical leak testing, chemical reactions (Angluin and Chen, 2008), electrical short detection, codes, multi-access channel communications (Biglieri and Gyrfi, 2007), molecular biology, VLSI testing, AIDS screening, whole-genome shotgun sequencing (Alon et al., 2002), DNA physical mapping (Grebinski and Kucherov, 1998) and game theory (Pelc, 2002). For a list of many other applications, see Du and Hwang (2000); Ngo and Du (2000); Bonis et al. (2005); Du and Hwang (2006); Cicalese (2013); Biglieri and Gyrfi (2007). Many of the new applications present new models and new problems. One of these is programming by example (PBE), a popular setting of program synthesis (Polozov and Gulwani, 2015; Barowy et al., 2015; Le and Gulwani, 2014; Gulwani, 2011; Jha et al., 2010). PBE has gained popularity because it enables end users to describe their intent to a program synthesizer via the intuitive means of input–output examples. The common setting of PBE is to synthesize a program based on a typically small set of user-provided examples, which are often an under-specification of the target program (Polozov and Gulwani, 2015; Barowy et al., 2015; Le and Gulwani, 2014; Gulwani, 2011). As a result, the synthesized program is not guaranteed to fully capture the user's intent. Another (less popular) PBE approach is to limit the program space to a

charco∋ 2017 N.H. Bshouty, D. Drachsler-Cohen, M. Vechev & E. Yahav.

small (finite) set of programs and ask the user membership queries while there are non-equivalent programs in the search space (Jha et al., 2010). A natural question is whether one can do better than the latter approach without sacrificing the correctness guaranteed by the former approach. In this paper, we answer this question for a class of specifications (i.e., formulas) that captures a wide range of programs.

We study the problem of learning a disjunctive (or dually, a conjunctive) formula describing the user intent through membership queries. To capture a wide range of program specifications, the formulas are over arbitrary, predefined predicates. In our setting, the end user is the teacher that can answer membership queries. This work enables PBE synthesizers to guarantee to the user that they have synthesized the correct program, while bounding the number of membership queries they pose, thus reducing the burden on the user.

More formally, let $\mathcal{F}$ be a set of predicates (i.e., boolean functions). Our goal is to learn the class $\mathcal{F}_\vee$ of any disjunction of predicates in $\mathcal{F}$. We present a learning algorithm SPEX, which learns any function in $\mathcal{F}_\vee$ with polynomially many queries. We then show that given some computational complexity conditions on the set of predicates, SPEX runs in polynomial time.

We demonstrate the above on two classes. The first is the class of disjunctions (or conjunctions, whose learning is the dual problem) over any set H of halfspaces over a constant dimension. For this class, we show that SPEX runs in polynomial time. In particular, this shows that learning any convex polytope over a constant dimension when the sides are from a given set H can be done in polynomial time. For the case where the dimension is not constant, we show that learning this class implies P=NP. We note that there are other applications for learning halfspaces; for example, Hegedűs (1995); Zolotykh and Shevchenko (1995); Abboud et al. (1999); Abasi et al. (2014).

The second class we consider is conjunctions over $\mathcal{F}$, where $\mathcal{F}$ is the set of variable inequalities, i.e., predicates of the form $[x_i > x_j]$ over n variables. If the set is acyclic ($\wedge\mathcal{F} \neq 0$), we show that learning can be done in polynomial time. If the set is cyclic ($\wedge\mathcal{F} = 0$), we show that learning is equivalent to the problem of enumerating all the maximal acyclic subgraphs of a directed graph, which is still an open problem (Acua et al., 2012; Borassi et al., 2013; Wasa, 2016).

The second class has practical importance because it consists of formulas that can be used to describe time-series charts. Time charts are used in many domains including financial analysis (Bulkowski, 2005), medicine (Chuah and Fu, 2007), and seismology (Morales-Esteban et al., 2010). Experts use these charts to predict important events (e.g., trend changes in a stock price) by looking for *patterns* in the charts. A lot of research has focused on common patterns and many platforms enable these experts to write a program that upon detecting a specific pattern alerts the user (e.g., some platforms for finance analysts are MetaTrader, MetaStock, Amibroker). Unfortunately, writing programs is a complex task for these experts, as they are not programmers. To help such experts, we integrated SPEX in a synthesizer that interacts with a user to learn a pattern (i.e., a conjunctive formula over variable inequalities). SPEX enables the synthesizer to guarantee that the synthesized program captures the user intent, while interacting with him only through membership queries that are visualized in charts.

The paper is organized as follows. Section 2 describes the model and class we consider. Section 3 provides the main definitions we require for the algorithm. Section 4 presents the SPEX algorithm, discusses its complexity, and describes conditions under which SPEX is polynomial. Sections 5 and 6 discuss the two classes we consider: halfspaces and variable inequalities. Finally, Section 7 shows the practical application of SPEX in program synthesis.

2. The Model and Class

Let $\mathcal{F}$ be a finite set of boolean functions over a domain X (possibly infinite). We consider the class of functions $\mathcal{F}_\vee := \{\vee_{f\in S} f \mid S \subseteq \mathcal{F}\}$. Our model assumes a *teacher* that has a *target function* $F \in \mathcal{F}_\vee$ and a *learner* that knows $\mathcal{F}$ but not the target function. The teacher can answer *membership queries* for the target function – that is, given $x \in X$ (from the learner), the teacher returns $F(x)$. The goal of the learner (the learning algorithm) is to find the target function with a minimum number of membership queries.

Notations Following are a few notations used throughout the paper. $\mathrm{OPT}(\mathcal{F}_\vee)$ denotes the minimum worst case number of membership queries required to learn a function F in $\mathcal{F}_\vee$. Given $F \in \mathcal{F}_\vee$, we denote by $S(F)$ the set that consists of all the functions in F. Formally, we define $S(F) = S$, where S is the unique subset of $\mathcal{F}$ such that $F = \bigvee_{f\in S} f$. For example, $S(f_1 \vee f_2) = \{f_1, f_2\}$. From this, it immediately follows that $F_1 \equiv F_2$ if and only if $S(F_1) = S(F_2)$. For a set of functions $S \subseteq \mathcal{F}$, we denote $\vee S := \vee_{f\in S} f$. Lastly, $[\mathcal{S}(x)]$ denotes the boolean value of a logical statement. Namely, given a statement $\mathcal{S}(x) : X \to \{T, F\}$ with a free variable x, its boolean function $[\mathcal{S}(x)] : X \to \{0, 1\}$ is defined as $[\mathcal{S}(x)] = 1$ if $\mathcal{S}(x) = T$, and $[\mathcal{S}(x)] = 0$ otherwise. For example, $[x \geq 2] = 1$ if and only if the interpretation of x is greater than 2.

3. Definitions and Preliminary Results

In this section, we provide the definitions used in this paper and show preliminary results. We begin by defining an equivalence relation over the set of disjunctions and defining the representatives of the equivalence classes. Thereafter, we define a partial order over the disjunctions and related notions (descendant, ascendant, and lowest/greatest common descendant/ascendant). We complete this section with the notion of a witness, which is central to our algorithm.

3.1. An Equivalence Relation Over $\mathcal{F}_\vee$

In this section, we present an equivalence relation over $\mathcal{F}_\vee$ and define the representatives of the equivalence classes. This enables us in later sections to focus on the representative elements from $\mathcal{F}_\vee$. Let $\mathcal{F}$ be a set of boolean functions over the domain X. The equivalence relation $=$ over $\mathcal{F}_\vee$ is defined as follows: two disjunctions $F_1, F_2 \in \mathcal{F}_\vee$ are equivalent ($F_1 = F_2$) if F_1 is logically equal to F_2. In other words, they represent the same function (from X to $\{0, 1\}$). We write $F_1 \equiv F_2$ to denote that F_1 and F_2 are identical; that is, they have the same representation. For example, consider $f_1, f_2 : \{0, 1\} \to \{0, 1\}$ where $f_1(x) = 1$ and $f_2(x) = x$. Then, $f_1 \vee f_2 = f_1$ but $f_1 \vee f_2 \not\equiv f_1$.

We denote by $\mathcal{F}^*_\vee$ the set of equivalence classes of $=$ and write each equivalence class as $[F]$, where $F \in \mathcal{F}_\vee$. Notice that if $[F_1] = [F_2]$, then $[F_1 \vee F_2] = [F_1] = [F_2]$. Therefore, for every $[F]$, we can choose the *representative element* to be $G_F := \vee_{F'\in S} F'$ where $S \subseteq \mathcal{F}$ is the maximum size set that satisfies $\vee S = F$. We denote by $G(\mathcal{F}_\vee)$ the set of all representative elements. Accordingly, $G(\mathcal{F}_\vee) = \{G_F \mid F \in \mathcal{F}_\vee\}$. As an example, consider the set $\mathcal{F}$ consisting of four functions $f_{11}, f_{12}, f_{21}, f_{22} : \{1, 2\}^2 \to \{0, 1\}$ where $f_{ij}(x_1, x_2) = [x_i \geq j]$. There are $2^4 = 16$ elements in $\mathrm{Ray}_2^2 := \mathcal{F}_\vee$ and five representative functions in $G(\mathcal{F}_\vee)$: $G(\mathcal{F}_\vee) = \{f_{11} \vee f_{12} \vee f_{21} \vee f_{22}, f_{12} \vee f_{22}, f_{12}, f_{22}, 0\}$ (where 0 is the zero function).

The below listed facts follow immediately from the above definitions:

Lemma 1 *Let $\mathcal{F}$ be a set of boolean functions. Then,*

1. *The number of logically non-equivalent boolean functions in $\mathcal{F}_\vee$ is $|G(\mathcal{F}_\vee)|$.*
2. *For every $F \in \mathcal{F}_\vee$, $G_F = F$.*
3. *For every $G \in G(\mathcal{F}_\vee)$ and $f \in \mathcal{F}\backslash S(G)$, $G \vee f \neq G$.*
4. *For every $F \in \mathcal{F}_\vee$, $\vee S(F) \equiv F$.*
5. *If $G_1, G_2 \in G(\mathcal{F}_\vee)$, then $G_1 = G_2$ if and only if $G_1 \equiv G_2$.*

3.2. A Partial Order Over $\mathcal{F}_\vee$

In this section, we define a partial order over $\mathcal{F}_\vee$ and present related definitions. The partial order, denoted by $\Rightarrow$, is defined as follows: $F_1 \Rightarrow F_2$ if F_1 logically implies F_2. Consider the Hasse diagram $H(\mathcal{F}_\vee)$ of $G(\mathcal{F}_\vee)$ for this partial order. The maximum (top) element in the diagram is $G_{\max} := \vee_{f\in\mathcal{F}} f$. The minimum (bottom) element is $G_{\min} := \vee_{f\in\emptyset} f$, i.e., the zero function. Figure 4 shows an illustration of the Hasse diagram of Ray_2^2 (from Section 3.1). Figures 5 and 6 show other examples of Hasse diagrams: Figure 5 shows the Hasse diagram of boolean variables, while Figure 6 shows an example that extends the example of Ray_2^2.

In a Hasse diagram, G_1 is a *descendant* (resp., *ascendent*) of G_2 if there is a (nonempty) downward path from G_2 to G_1 (resp., from G_1 to G_2), i.e., $G_1 \Rightarrow G_2$ (resp., $G_2 \Rightarrow G_1$) and $G_1 \neq G_2$. G_1 is an *immediate descendant* of G_2 in $H(\mathcal{F}_\vee)$ if $G_1 \Rightarrow G_2$, $G_1 \neq G_2$ and there is no G such that $G \neq G_1$, $G \neq G_2$ and $G_1 \Rightarrow G \Rightarrow G_2$. G_1 is an *immediate ascendant* of G_2 if G_2 is an immediate descendant of G_1. We now show (all proofs for this section appear in Appendix B):

Lemma 2 *Let G_1 be an immediate descendant of G_2 and $F \in \mathcal{F}_\vee$. If $G_1 \Rightarrow F \Rightarrow G_2$, then $G_1 = F$ or $G_2 = F$.*

We denote by $\mathrm{De}(G)$ and $\mathrm{As}(G)$ the sets of all the immediate descendants and immediate ascendants of G, respectively. We further denote by $\mathrm{DE}(G)$ and $\mathrm{AS}(G)$ the sets of all G's descendants and ascendants, respectively. For G_1 and G_2, we define their *lowest common ascendent* (resp., greatest common descendant) $G = \mathrm{lca}(G_1, G_2)$ (resp., $G = \mathrm{gcd}(G_1, G_2)$) to be the boolean function $G \in G(\mathcal{F}_\vee)$ – that is, the minimum (resp., maximum) element in $\mathrm{AS}(G_1) \cap \mathrm{AS}(G_2)$ (resp., $\mathrm{DE}(G_1) \cap \mathrm{DE}(G_2)$). Therefore, we can show Lemma 3. Lemma 3 abbreviates ($G_1 \Rightarrow G$ and $G_2 \Rightarrow G$) to $G_1, G_2 \Rightarrow G$ and ($G \Rightarrow G_1$ and $G \Rightarrow G_2$) to $G \Rightarrow G_1, G_2$.

Lemma 3 *Let $G_1, G_2 \in G(\mathcal{F}_\vee)$ and $F \in \mathcal{F}_\vee$.*

1. *If $G_1, G_2 \Rightarrow F \Rightarrow \mathrm{lca}(G_1, G_2)$, then $F = \mathrm{lca}(G_1, G_2)$.*
2. *If $\mathrm{gcd}(G_1, G_2) \Rightarrow F \Rightarrow G_1, G_2$, then $F = \mathrm{gcd}(G_1, G_2)$.*

Lemma 3 leads us to Lemma 4:

Lemma 4 *Let $G_1, G_2 \in G(\mathcal{F}_\vee)$. Then, $\mathrm{lca}(G_1, G_2) = G_1 \vee G_2$.*
In particular, if G_1, G_2 are two distinct immediate descendants of G, then $G_1 \vee G_2 = G$.

Note that this does not imply that $S(G_1 \vee G_2) = S(G_1) \cup S(G_2) = S(\mathrm{lca}(G_1, G_2))$. In particular, $G_1 \vee G_2$ is not necessarily in $G(\mathcal{F}_\vee)$; see, for example, Figure 5 (right).

Lemma 5 follows from the fact that if G_1 is a descendant of G_2, then $G_1 \Rightarrow G_2$, and therefore, $G_1 \vee G_2 = G_2$.

Lemma 5 *If G_1 is a descendant of G_2, then $S(G_1) \subsetneq S(G_2)$.*

Lemma 5 enables us to show the following.

Lemma 6 *Let $G_1, G_2 \in G(\mathcal{F}_\vee)$. Then, $S(G_1) \cap S(G_2) = S(\gcd(G_1, G_2))$.*
In particular, if $G_1, G_2 \in G(\mathcal{F}_\vee)$, then $\vee(S(G_1) \cap S(G_2)) \in G(\mathcal{F}_\vee)$.
Also, if G_1, G_2 are two distinct immediate ascendants of G, then $S(G_1) \cap S(G_2) = S(G)$.

Note that this does not imply that $G_1 \wedge G_2 = \gcd(G_1, G_2)$; see, for example, Figure 5 (right).

3.3. Witnesses

Finally, we define the term *witness*. Let G_1 and G_2 be elements in $G(\mathcal{F}_\vee)$. An element $a \in X$ is a *witness* for G_1 and G_2 if $G_1(a) \neq G_2(a)$. We now show two central lemmas.

Lemma 7 *Let G_1 be an immediate descendant of G_2. If $a \in X$ is a witness for G_1 and G_2, then:*

1. *$G_1(a) = 0$ and $G_2(a) = 1$.*
2. *For every $f \in S(G_1)$, $f(a) = 0$.*
3. *For every $f \in S(G_2) \backslash S(G_1)$, $f(a) = 1$.*

Proof Since $G_1 \Rightarrow G_2$, it must be that $G_2(a) = 1$ and $G_1(a) = 0$. Namely, for every $f \in S(G_1)$, $f(a) = 0$. Let $f \in S(G_2) \backslash S(G_1)$. Consider $F = G_1 \vee f$. By bullet *3* in Lemma 1, $F \neq G_1$. Since $G_1 \Rightarrow F \Rightarrow G_2$, by Lemma 2, $F = G_2$. Therefore, $f(a) = G_1(a) \vee f(a) = F(a) = G_2(a) = 1$. ■

Lemma 8 *Let $\mathrm{De}(G) = \{G_1, G_2, \ldots, G_t\}$ be the set of immediate descendants of G. If a is a witness for G_1 and G, then a is not a witness for G_i and G for all $i > 1$. That is, $G_1(a) = 0$, $G(a) = 1$, and $G_2(1) = \cdots = G_t(a) = 1$.*

Proof By Lemma 7, $G(a) = 1$ and $G_1(a) = 0$. By Lemma 4, for any G_i, $i \geq 2$, we have $G = G_1 \vee G_i$. Therefore, $1 = G(a) = G_1(a) \vee G_i(a) = G_i(a)$. ■

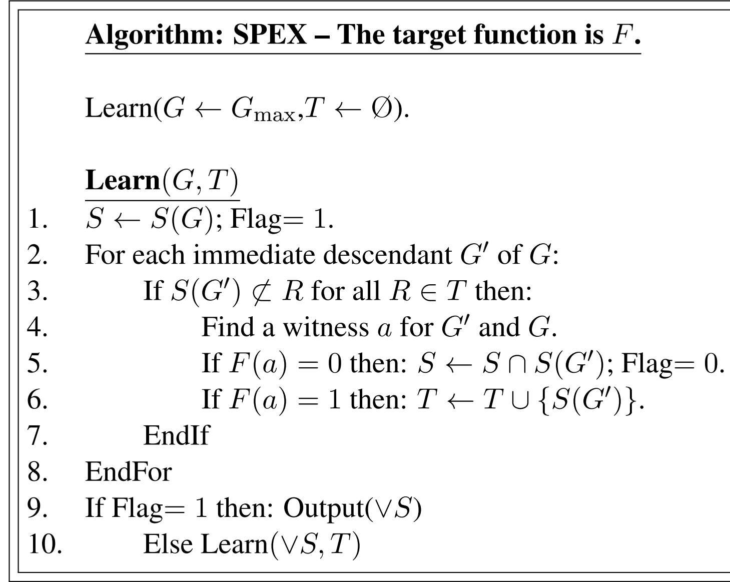
4. The Algorithm

In this section, we present our algorithm to learn a target disjunction over $\mathcal{F}$, called SPEX (short for *specifications from examples*). Our algorithm relies on the following results.

Lemma 9 *Let G' be an immediate descendant of G, $a \in X$ be a witness for G and G', and G'' be a descendant of G.*

1. *If $G''(a)$=0, G'' is a descendant of G' or equal to G'. In particular, $S(G'') \subseteq S(G')$.*
2. *If $G''(a)$=1, G'' is neither a descendant of G' nor equal to G'. In particular, $S(G'') \not\subseteq S(G')$.*

Proof Since G'' is a descendant of G, we have $S(G'') \subsetneq S(G)$. By Lemma 7, for every $f \in S(G')$, we have $f(a) = 0$, and for every $f \in S(G) \backslash S(G')$, we have $f(a) = 1$. Thus, if $G''(a) = 0$, then no $f \in S(G) \backslash S(G')$ is in $S(G'')$ (otherwise, $G''(a) = 1$). Therefore, $S(G'') \subseteq S(G')$ and G'' is a descendant of G' or equal to G'. Otherwise, if $G''(a) = 1$, then since $G'(a) = 0$, for every descendant G_0 of G' we have $G_0(a) = 0$ and thus G'' is neither a descendant of G' nor equal to G'.■



Algorithm: SPEX – The target function is F.

Learn($G \leftarrow G_{\max}$,$T \leftarrow \emptyset$).

Learn(G, T)

1. $S \leftarrow S(G)$; Flag$= 1$.
2. For each immediate descendant G' of G:
3. If $S(G') \not\subset R$ for all $R \in T$ then:
4. Find a witness a for G' and G.
5. If $F(a) = 0$ then: $S \leftarrow S \cap S(G')$; Flag$= 0$.
6. If $F(a) = 1$ then: $T \leftarrow T \cup \{S(G')\}$.
7. EndIf
8. EndFor
9. If Flag$= 1$ then: Output($\vee S$)
10. Else Learn($\vee S, T$)

Figure 1: The SPEX algorithm for learning functions in $\mathcal{F}_\vee$.

Lemma 9 drives the operation of SPEX. To find F (more precisely, G_F), SPEX starts from the maximal element in $G(\mathcal{F}_\vee)$ and traverses downwards through the Hasse diagram. At each step, SPEX considers an element G, checks its witnesses against its immediate descendants, and poses a membership query for each. If F and G agree on the witness of G and G', then by Lemma 9, F cannot be G' or its descendant, and thus these are pruned from the search space. Otherwise, if F and G' agree on the witness, then F must be G' or its descendant, and thus all other elements in $G(F_\vee)$ are pruned.

The SPEX algorithm is depicted in Figure 1. SPEX calls the recursive algorithm Learn, which takes a candidate G and a set of subsets of $\mathcal{F}$, T, which stores the elements already eliminated from $\mathcal{F}_\vee$. Learn also relies on S, a set of functions over which F (i.e., G_F) is defined (i.e., $S(G_F) \subseteq S$). During the execution, S may be reduced. If not, then $\vee S = F$. Learn begins by initializing S to $S(G)$. Then, it examines the immediate descendants of G whose ancestors have not been eliminated. When considering G', a witness a is obtained and Learn poses a membership query to learn $F(a)$. If $F(a) = 0$ (recall that $G(a) = 1$ since a is a witness), then $G \neq F$ and F is inferred to be a descendant of G' and is thus over the functions in $S(G')$. Accordingly, S is reduced. Otherwise, F is not a descendant of G', and G' and its descendants are eliminated from the search space by adding $S(G')$ to T. Finally, if G and F agreed on all the witnesses (evident by the Flag variable), then $\vee S$ is returned (since $G = F$). Otherwise, Learn is invoked on $\vee S$ and T.

Theorem 10 *If the witnesses and the descendants of any G can be found in time t, then SPEX (Algorithm 1) learns the target function in time $t \cdot |\mathcal{F}|$ and at most $|\mathcal{F}| \cdot \max_{G \in G(\mathcal{F}_\vee)} |\mathrm{De}(G)|$ membership queries.*

The complexity proof follows from the following arguments. First, every invocation of SPEX presents a membership query for every immediate descendant of G, and thus the number of membership queries of a single invocation is at most the maximal number of immediate descendants,

$\max_{G\in G(\mathcal{F}_\vee)}|\mathrm{De}(G)|$. Second, recursive invocations always consider a descendant of the currently inspected candidate. Thus, the recursion depth is bounded by the height of the Hasse diagram, $|\mathcal{F}|$. This implies the total bound of $|\mathcal{F}|\cdot\max_{G\in G(\mathcal{F}_\vee)}|\mathrm{De}(G)|$ membership queries. The fact that SPEX learns the target function follows from Lemma 11.

Lemma 11 *Let F be the target function. If Learn returns $\vee S$, then $G_F = \vee S$ ($\star$). Otherwise, if Learn$(G_{\max}, \emptyset)$ calls Learn$(\vee S, T)$, then:*

1. *$S(G_F) \subseteq S$. That is, G_F is a descendant of $\vee S$ or equal to $\vee S$.*
2. *$S(G_F) \not\subseteq R$ for all $R \in T$. That is, G_F is not a descendant of any $\vee R$, for $R \in T$ or equal to $\vee R$.*

Proof The proof is by induction. Obviously, the induction hypothesis is true for $(G_{\max}, \emptyset)$. Assume the induction hypothesis is true for $(\vee S, T)$. That is, $S(G_F) \subseteq S$ and $S(G_F) \not\subseteq R$ for all $R \in T$. Let $G'_1, \dots, G'_\ell$ be all the immediate descendants of $\vee S$. If $S(G'_i) \subseteq R$ for some $R \in T$, G'_i and all its descendants G'' satisfy $S(G'') \subseteq S(G'_i) \subseteq R$ and thus G_F is not G'_i or a descendant of G'_i.

Assume now that $S(G'_i) \not\subseteq R$ for all $R \in T$. Let $a^{(i)}$ be a witness for $\vee S$ and G'_i. If $F(a^{(i)}) = 1$, then by Lemma 9 G_F is not a descendant of G'_i and not equal to G'_i. This implies that $S(G_F) \not\subseteq S(G'_i)$, which is why $S(G'_i)$ is added to T (Line 6 in the Algorithm). This proves bullet 2.

If $F(a^{(i)}) = 1$ for all i, then $G_F = \vee S$. This follows since by Lemma 9, F is not any of $\vee S$ descendants; thus by the induction hypothesis, it must be $\vee S$. This is the case when the Flag variable does not change to 0 and the algorithm outputs $\vee S$. This proves ($\star$).

If $F(a^{(i)}) = 0$, then by Lemma 9, G_F is a descendant of G'_i or equal to G'_i. Let I be the set of all indices i for which $F(a^{(i)}) = 0$. Then, G_F is a descendant of (or equal to) all G'_i, $i \in I$, and therefore, G_F is a descendant of or equal to $\gcd(\{G'_i\}_{i\in I})$. By Lemma 6, $S(\gcd(\{G'_i\}_{i\in I})) = \cap_{i\in I} S(G_i)$. Thus, the algorithm in Line 5 takes the new S to be $\cap_{i\in I} S(G_i)$. This proves bullet 1. ■

4.1. Lower Bound

The number of different boolean functions in $\mathcal{F}_\vee$ is $|G(\mathcal{F}_\vee)|$, and therefore, from the information theoretic lower bound we get: $\mathrm{OPT}(\mathcal{F}_\vee) \geq \lceil \log |G(\mathcal{F}_\vee)| \rceil$. We now prove the lower bound.

Theorem 12 *Any learning algorithm that learns $\mathcal{F}_\vee$ must ask at least $\max(\log|G(\mathcal{F}_\vee)|, \max_{G\in G(\mathcal{F}_\vee)}|\mathrm{De}(G)|)$ membership queries. In particular, SPEX (Algorithm 1) asks at most $|\mathcal{F}|\cdot\mathrm{OPT}(\mathcal{F}_\vee)$ membership queries.*

Proof Let G' be such that $m = |\mathrm{De}(G')| = \max_{G\in G(\mathcal{F}_\vee)}|\mathrm{De}(G)|$. Let $G_1, \dots, G_m$ be the immediate descendants of G'. If the target function is either G' or one of its immediate descendants, then any learning algorithm must ask a membership query $a^{(i)}$ such that $G'(a^{(i)}) = 1$ and $G_i(a^{(i)}) = 0$. Without such an assignment, the algorithm cannot distinguish between G' and G_i. By Lemma 8, $a^{(i)}$ is a witness only to G_i, and therefore, we need at least m membership queries. ■

4.2. Finding All Immediate Descendants of G

A missing detail in our algorithm is how to find the immediate descendants of G in the Hasse diagram $H(S(G))$. In this section, we explain how to obtain them. We first characterize the elements

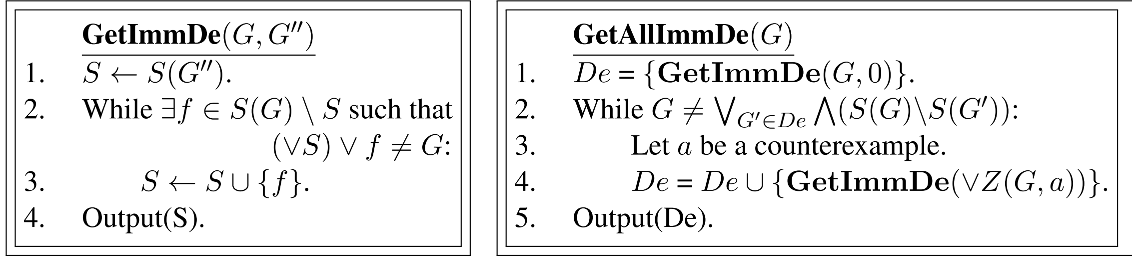


Figure 2: Left: the GetImmDe operation. Right: the GetAllImmDe operation.

in $H(S(G))$ (compared to the other elements in $\mathcal{F}_\vee$), which is necessary because the immediate descendants are part of $H(S(G))$. We then give a characterization of the immediate descendants (compared to other descendants), which leads to an operation that computes an immediate descendant from a descendant. We finally show how to compute descendants that lead to obtaining different immediate descendants. This completes the description of how SPEX can obtain all immediate descendants.

By the definition of a representative, for every $F \in \mathcal{F}_\vee$, $G_F = \vee_{f \Rightarrow F} f$. To decide whether $F \in \mathcal{F}_\vee$ is a representative, i.e., whether $F \in G(\mathcal{F}_\vee)$, we use Lemma 13 (whose proof directly follows from the definition of $G(\mathcal{F}_\vee)$).

Lemma 13 *Let $F \in \mathcal{F}_\vee$. $F \in G(\mathcal{F}_\vee)$ if and only if for every $f \in \mathcal{F} \backslash S(F)$ we have $F \vee f \neq F$.*

Lemma 14 shows how to decide whether G' is an immediate descendant of G.

Lemma 14 *Let $G, G' \in G(\mathcal{F}_\vee)$. G' is an immediate descendant of G if and only if $G' \neq G$, $S(G') \subset S(G)$ and for every $f \in S(G) \backslash S(G')$ we have $G' \vee f = G$.*

If $G' \neq G$, $S(G') \subset S(G)$ and for some $f \in S(G) \backslash S(G')$ we have $G' \vee f \neq G$, then $G_{G' \vee f}$ is a descendant of G and an ascendant of G'.

Proof *Only if:* Let G' be an immediate descendant of G, i.e., $G' \neq G$, $G' \Rightarrow G$ and $S(G') \subset S(G)$. Let $f \in S(G) \backslash S(G')$. Since $G' \Rightarrow (G' \vee f) \Rightarrow G$ and $G' \neq G' \vee f$, we get $G' \vee f = G$.

If: Suppose $G' \neq G$, $G' \Rightarrow G$ and for every $f \in S(G) \backslash S(G')$, we have $G' \vee f = G$. If G' is not an immediate descendant of G, then let G'' be a descendant of G and an immediate ascendant of G''. Take any $f \in S(G'') \backslash S(G') \subset S(G) \backslash S(G')$. Then, $G' \vee f = G'' \neq G$ – a contradiction. This also proves the last statement of Lemma 14. ■

Lemma 14 shows how to compute an immediate descendant from a descendant, which we phrase in an operation called GetImmDe (Figure 2, left). GetImmDe takes G and a descendant G'' of G (which can even be the zero function), initializes $S = S(G'')$, and as long as possible, repeatedly extends S as follows: for $f \in S(G) \backslash S$ if $(\vee S) \vee f \neq G$, f is added to S.

GetImmDe can be used to obtain the first immediate descendant by calling it with $G'' = 0$. We next show how to obtain a descendant for which GetImmDe will return a different immediate descendant. To this end, we define the following: For $G \in G(\mathcal{F}_\vee)$ and a set $X' \subseteq X$, $Z(G, X') = \{f \in S(G) \mid f(X') = 0\}$, where $f(X') = \vee_{x \in X'} f(x)$. When $X' = \{x\}$, we abbreviate to $Z(G, x)$. Obviously,

$$Z(G, X') = \bigcap_{x \in X'} Z(G, x). \tag{1}$$

Lemma 15 relates this new definition to the descendants of G.

Lemma 15 *Let $G \in G(\mathcal{F}_\vee)$ and $X' \subseteq G^{-1}(1)$ be a nonempty set. Then, $G' = \vee Z(G, X') \in G(\mathcal{F}_\vee)$ and G' is a descendant of G.*

For every immediate descendant G' of G, there is $X' \subseteq G^{-1}(1)$ such that $\vee Z(G, X') = G'$.

Proof First notice that $G'(X') = 0$. Suppose on the contrary that $G' \notin G(\mathcal{F}_\vee)$. Then, there is $f \in S(G)\backslash Z(G, X')$ such that $G' \vee f = G'$. Since $f \notin Z(G, X')$, there is $z \in X'$ such that $f(z) = 1$ and then $G'(X') = (G' \vee f)(X') \neq 0$ – a contradiction. Therefore, $G' \in G(\mathcal{F}_\vee)$. By the definition of Z, $S(G') \subseteq S(G)$ and thus G' is a descendant of G.

Let G' be an immediate descendant of G and let $X' = \{x \in X \mid G'(x) = 0 \text{ and } G(x) = 1\}$. Then, $X' \subseteq G^{-1}(1)$. We now show $Z(G, X') = S(G')$. $S(G') \subseteq Z(G, X')$ because if $f \in S(G')$, then for all $x \in X'$, $f(x) = 0$ and thus $f \in Z(G, X')$. We next prove that $Z(G, X') \subseteq S(G')$. Let $f \in Z(G, X')$ and x_0 be a witness for G and G', i.e., $G(x_0) = 1$ and $G'(x_0) = 0$. Therefore, $x_0 \in X'$ and $f(x_0) = 0$ by the definition of Z. By Lemma 7, for every $f \in S(G)\backslash S(G')$, we have $f(a) = 1$. Since $f \in S(G)$, it must be that $f \in S(G')$. ■

Lemma 15 shows how to construct descendants from elements in X. Lemma 16 determines when all immediate descendants of G were obtained, and if not, how to obtain a new element in X that leads to a new immediate descendant. The algorithm that finds all immediate descendants (Figure 2, right) follows directly from this lemma. In the following we denote $\bigwedge S = \bigwedge_{f \in S} f$.

Lemma 16 *Let $G_1, \ldots, G_m$ be immediate descendants of G. There is no other immediate descendant for G if and only if*

$$G = \bigvee_{i=1}^{m} \bigwedge (S(G)\backslash S(G_i)). \tag{2}$$

If (2) does not hold, then for any counterexample a for (2), we have $\vee Z(G, a)$ is a descendant of G but not equal to and not a descendant of any G_i, $i = 1, \ldots, m$.

Proof *Only if:* Suppose $G \neq \vee_{i=1}^m \wedge (S(G)\backslash S(G_i))$ and let a be a counterexample. Since for all i, $S(G)\backslash S(G_i) \subseteq S(G)$, we have $\vee_{i=1}^m \wedge (S(G)\backslash S(G_i)) \Rightarrow G$. Therefore, $G(a) = 1$ and for every i there is $f_i \in S(G)\backslash S(G_i)$ such that $f_i(a) = 0$. Consider $G' = Z(G, a)$. Since $f_i(a) = 0$, we have $f_i \in S(G')$. Since $f_i \notin S(G_i)$, G' is not a descendant of G_i. Since G' is a descendant of G and not a descendant of any G_i, there must be another immediate descendant of G.

If: Denote $W = \bigvee_{i=1}^m \bigwedge (S(G)\backslash S(G_i))$. Let G' be another immediate descendant of G. Let a be a witness for G and G'. Then $G(a) = 1$ and, by Lemma 7, for every $f \in S(G)\backslash S(G')$, we have $f(a) = 1$ and for every $f \in S(G')$, we have $f(a) = 0$. Since $S(G') \not\subset S(G_i)$, we have $S(G)\backslash S(G_i) \not\subset S(G)\backslash S(G')$, and therefore, $(S(G)\backslash S(G_i)) \cap S(G')$ is not empty. Choose $f_i \in (S(G)\backslash S(G_i)) \cap S(G')$. Then, $f_i \in S(G)\backslash S(G_i)$ and since $f_i \in S(G')$, $f_i(a) = 0$. Therefore, $W(a) = 0$. Since $G(a) = 1$, we get $G \neq W$. ■

4.3. Critical Points

In this section, we show that if one can find certain points (the *critical points*), then the immediate descendants can be computed in polynomial time (in the number of these points) and thus SPEX runs

in polynomial time. In particular, if the number of points is polynomial and they can be obtained in polynomial time, then SPEX runs in polynomial time.

A set of points $C \subseteq X$ is called *a critical point set* for $\mathcal{F}$ if for every $S \subseteq \mathcal{F}$ and if $H = \bigwedge_{f\in S} f \wedge \bigwedge_{f\in\mathcal{F}\backslash S} \bar{f} \neq 0$, then there is a point in $c \in C$ such that $H(c) = 1$.

We now show how to use the critical points to find the immediate descendants.

Lemma 17 *Let $C \subseteq X$ be a set of points. If C is a set of critical points for $\mathcal{F}$, then all the immediate descendants of $G \in G(\mathcal{F}_\vee)$ can be found in time $|C| \cdot |S(G)|$.*

Proof Let $G_1, \dots, G_m$ be some of the immediate descendants of G. To find another immediate descendant we look for a point a such that $G(a) = 1$, and for every i, there is $f_i \in S(G)\backslash S(G_i)$ such that $f_i(a) = 0$. Let a be such point. Consider $S = \{f \in \mathcal{F} \mid f(a) = 1\}$ and let $H = \bigwedge_{f\in S} f \wedge \bigwedge_{f\in\mathcal{F}\backslash S} \bar{f}$. Since $H(a) \neq 0$, there is a critical point $b \in C$ such that $H(b) = 1$. By the definition of b, $G(b) = 1$ and for every i there is $f_i \in S(G)\backslash S(G_i)$ such that $f_i(b) = 0$. Therefore, b can be used to find a new descendant of G. To find the descendants we need, in the worst case, to substitute all the assignments of C in all the descendants $G_1, \dots, G_m$ and G. For all the descendants, this takes at most $|C| \cdot |S(G)|$ steps, which implies the time complexity. ■

We now show how to generate the set of critical points.

Lemma 18 *If for every $S, R \subseteq \mathcal{F}$ one can decide whether $H_{S,R} = \bigwedge_{f\in S} f \wedge \bigwedge_{f\in R} \bar{f} \neq 0$ in time T and if so, find $a \in X$ such that $H_{R,S}(a) = 1$, then a set of critical points C can be found in time $|C| \cdot T \cdot |\mathcal{F}|$.*

Proof The set is constructed inductively, in stages. Let $\mathcal{F} = \{f_1, \dots, f_t\}$ and denote $\mathcal{F}_i = \{f_1, \dots, f_i\}$. Suppose we have a set $K_i = \{S \subseteq [i] \mid \bigwedge_{f\in S} f \wedge \bigwedge_{f\in[i]\backslash S} \bar{f} \neq 0\}$; then we define $K_{i+1} = \{g \wedge f_{i+1} \mid g \in K_i \text{ and } g \wedge f_{i+1} \neq 0\} \cup \{g \wedge \overline{f_{i+1}} \mid g \in K_i \text{ and } g \wedge \overline{f_{i+1}} \neq 0\}$. ■

5. A Polynomial Time Algorithm for Halfspaces in a Constant Dimension

In this section, we show two results. The first is that when $\mathcal{F}$ is a set of halfspaces over a constant dimension, one can find a polynomial-sized critical point set in polynomial time, and therefore, SPEX can run in polynomial time. We then show that unless $P = NP$, this result cannot be extended to non-constant dimensions.

A halfspace of dimension d is a boolean function of the form:

$$f(x_1, \dots, x_d) = [a_1x_1 + \cdots + a_dx_d \geq b] = \begin{cases} 1 & \text{if } a_1x_1 + \cdots + a_dx_d \geq b \\ 0 & \text{otherwise} \end{cases}$$

where $(x_1, \dots, x_d) \in \Re^d$ and $a_1, \dots, a_d, b$ are real numbers. Therefore, $f : \Re^d \to \{0, 1\}$.

We now prove that the set of critical points is of a polynomial size.

Lemma 19 *Let $\mathcal{F}$ be a set of halfspaces in dimension d. There is a set of critical points C for $\mathcal{F}$ of size $|\mathcal{F}|^{d+1}$.*

Proof Define the dual set of halfspaces. That is, for every $x \in \Re^d$, the dual function $x^\perp : \mathcal{F} \to \{0,1\}$ where $x^\perp(f) = f(x)$. It is well known that the VC-dimension of this set is at most $d+1$. By the Sauer-Shelah lemma, the result follows. ■

Next, we prove that the set of critical points can be computed in polynomial time.

Lemma 20 *Let $\mathcal{F}$ be a set of halfspaces in dimension d. A set of critical points C for $\mathcal{F}$ of size $|\mathcal{F}|^{d+1}$ can be found in time poly$(|\mathcal{F}|^d)$.*

Proof Follows from Lemma 18 and the fact that linear programming (required to check whether $g \wedge f_i \neq 0$) takes polynomial time. ■

By the above results, we conclude:

Theorem 21 *Let $\mathcal{F}$ be a set of halfspaces in dimension d. There is a learning algorithm for $\mathcal{F}_\vee$ that runs in time $|\mathcal{F}|^{O(d)}$ and asks at most $|\mathcal{F}| \cdot \mathrm{OPT}(\mathcal{F}_\vee)$ membership queries.*

In particular, when the dimension d is constant, the algorithm runs in polynomial time.

Next, we show that the above cannot be extended to a non-constant dimension:

Theorem 22 *If every set $\mathcal{F}$ of halfspaces deciding whether $F \in \mathcal{F}_\vee$ is a descendant of $\vee\mathcal{F}$ can be done in polynomial time, then $P = NP$.*

Proof The reduction is from the problem of dual 3SAT – that is, given the literals $\{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$ and the terms $T_1, \ldots, T_m$ where each T_i is a conjunction of three literals, decide whether $T_1 \vee \cdots \vee T_m = 1$.

Given the terms $T_1, \ldots, T_m$, each can be translated into a halfspace. For example, the term $x_1 \wedge \bar{x}_2 \wedge x_3$ corresponds to the halfspace $[x_1 + (1 - x_2) + x_3 \geq 3] = [x_1 - x_2 + x_3 \geq 2]$. Now, consider $\mathcal{F} = \{T_1, \ldots, T_m, 1\}$. Then $G_{\max} = 1$ and $T_1 \vee \ldots \vee T_m \neq 1$ if and only if $T_1 \vee \cdots \vee T_m$ is the only immediate descendant of $G_{\max}$.

■

6. Duality and a Polynomial Time Algorithm for Variable Inequality Predicates

In this section, we study the learnability of conjunctions over variable inequality predicates. In the acyclic case, we provide a polynomial time learning algorithm. In the general case, we show that the learning problem is equivalent to the open problem of enumerating all the maximal acyclic subgraphs of a given directed graph.

Consider the set of boolean functions $\mathcal{F}^I := \{[x_i > x_j] \mid (i,j) \in I\}$ for some $I \subseteq [n]^2$ where $[n] = \{1, 2, \ldots, n\}$ and the variables x_i are interpreted as real numbers. We define $[x_i > x_j] = 1$ if $x_i > x_j$; and 0 otherwise. We assume throughout this section that $(i,i) \notin I$ for all i.

We consider the dual class $\mathcal{F}_\wedge := \{\wedge_{f \in S} f \mid S \subseteq \mathcal{F}\}$. By duality (De Morgan's law), all our results are true for learning $\mathcal{F}_\wedge$ (after swapping $\vee$ with $\wedge$). The dual SPEX algorithm is depicted in Figure 3.

For a set $J \subseteq I$, we define $F_J = \wedge_{(i,j) \in J}[x_i > x_j]$. For $F \in \mathcal{F}^I_\wedge$ we define $\mathcal{I}(F) = \{(i,j) \mid [x_i > x_j] \text{ is in } F\}$. Note that $\mathcal{I}(F_J) = J$. For example, $\mathcal{I}([x_1 > x_2] \wedge [x_3 > x_1]) = \{(1,2), (3,1)\}$.

The *directed graph of* $I \subseteq [n]^2$ is $\mathcal{G}_I = ([n], I)$. The *reachability matrix* of I, denoted by $R(I)$, is an $n \times n$ matrix where $R(I)_{i,j} = 1$ if there is a (directed) path from i to j in $\mathcal{G}_I$; otherwise, $R(I)_{i,j} = 0$. We say that I is *acyclic* (resp., *cyclic*) if the graph $\mathcal{G}_I$ is acyclic (resp., cyclic). We say that an assignment to the variables $a \in [n]^n$ is a *topological sorting* of I if for every $(i, j) \in I$, we have $a_i > a_j$. It is known that I has a topological sorting if and only if I is acyclic. Also, it is known that a topological sorting for an acyclic set can be found in linear time (see Knuth (1997), Volume 1, Section 2.2.3 and Cormen et al. (2001)). Next, we study the learnability of $\mathcal{F}^I_\wedge$ when I is acyclic and following this, we discuss the general case.

6.1. Acyclic Sets

We now examine the case when I is acyclic. Here, the number of critical points of $\mathcal{F}^I$ where $I = \{(1, 2), (2, 3), \cdots, (n - 1, n)\}$ is 2^{n-1}. Therefore, using Lemma 17 does not enable us to obtain a polynomial time algorithm. Accordingly, we show a different way to determine whether a function is a representative (i.e., in $G(\mathcal{F}_\wedge)$), and then show how to obtain the immediate descendants in quadratic time (in n) and the witnesses in linear time. As a result, SPEX can run in polynomial time. Finally, we show that the number of membership queries is at most $|I|$.

Before we show the main lemma, Lemma 24, we present Lemma 23, a trivial lemma that we use to prove Lemma 24.

Lemma 23 *Let $I \subseteq [n]^2$ be an acyclic set, $F \in \mathcal{F}^I_\wedge$, and $a \in [n]^n$. Then, $F(a) = 1$ if and only if a is a topological sorting of $\mathcal{G}_{\mathcal{I}(F)}$.*

In particular, F is satisfiable and a satisfying assignment $a \in [n]^n$ can be found in linear time.

Next we show our main lemma that enables us to determine the representative elements and the immediate descendants (in Lemmas 25–27). The proof is provided in Appendix C.

Lemma 24 *Let I be acyclic and $F_1, F_2 \in \mathcal{F}^I_\wedge$. Then, $F_1 = F_2$ if and only if $R(\mathcal{I}(F_2))$=$R(\mathcal{I}(F_1))$.*

We now show how to decide whether $F \in G(\mathcal{F}^I_\wedge)$ – that is, whether F is a representative.

Lemma 25 *Let I be an acyclic set and $F \in \mathcal{F}^I_\wedge$. $F \in G(\mathcal{F}^I_\wedge)$ if and only if for every $(i, j) \in I \backslash \mathcal{I}(F)$ there is no path from i to j in $\mathcal{G}_{\mathcal{I}(F)}$.*

Proof *If:* If for every $(i, j) \in I\backslash\mathcal{I}(F)$ there is no path from i to j in $\mathcal{G}_{\mathcal{I}(F)}$, then for every $(i, j) \in I\backslash\mathcal{I}(F)$, $R(\mathcal{I}(F) \cup \{(i, j)\}) \neq R(\mathcal{I}(F))$. By Lemma 24, this implies that $F \wedge [x_i > x_j] \neq F$. By (the dual result of) Lemma 13, the result follows.

Only if: Now let $F \in G(\mathcal{F}^I_\wedge)$. By Lemma 13, for every $[x_i > x_j] \notin F$, we have $F \wedge [x_i > x_j] \neq F$. Therefore, there is an assignment a that satisfies $a_i \leq a_j$ and $F(a) = 1$. As before, if there is a path in $\mathcal{G}_{\mathcal{I}(F)}$ from i to j, then we get a contradiction. ■

We now show how to determine the immediate descendants of G in polynomial time.

Lemma 26 *Let I be acyclic. The immediate descendants of $G \in G(\mathcal{F}^I_\wedge)$ are all $G^{r,s} := F_{\mathcal{I}(G)\backslash\{(r,s)\}}$ where $(r, s) \in \mathcal{I}(G)$ and there is no path from r to s in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(r,s)\}}$.*

In particular, for all $G \in G(\mathcal{F}^I_\wedge)$, we have $|\mathrm{De}(G)| \leq |\mathcal{I}(G)| \leq |I|$.

The proof is in Appendix C. We now show how to find a witness.

Lemma 27 *Let I be acyclic, $G \in G(\mathcal{F}_\wedge^I)$, and $G^{r,s} := F_{\mathcal{I}(G)\backslash\{(r,s)\}}$ be an immediate descendant of G. A witness for G and $G^{r,s}$ can be found in linear time.*

Proof By Lemma 26, $(r,s) \in \mathcal{I}(G)$ and there is no path from r to s in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(r,s)\}}$. Therefore, if we match vertices r and s in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(r,s)\}}$ we get an acyclic graph $\mathcal{G}'$. Then, a topological sorting a for $\mathcal{G}'$ is a satisfying assignment for $G^{r,s}$ that satisfies $a_r = a_s$. Since $[x_r > x_s] \in S(G)$, we get $G(a) = 0$. Therefore, a is a witness for G and $G^{r,s}$. ■

To learn a function in $\mathcal{F}_\wedge^I$, SPEX needs to find the immediate descendants of G and a witness for each immediate descendant and G. By Lemma 26, this involves finding a path between every two nodes in the directed graph $\mathcal{G}_{I(G)}$, which can be done in polynomial time. By Lemma 27, to find a witness, SPEX needs a topological sorting, which can be done in linear time. Therefore, SPEX runs in polynomial time. Therefore, by Theorem 10 and Lemma 26, the class $\mathcal{F}_\wedge^I$ is learnable in polynomial time with at most $|I|^2$ membership queries. We now show that the number of membership queries is actually lower and equal to $|I|$.

Theorem 28 *Let $I \subseteq [n]^2$ be acyclic. The class $\mathcal{F}_\wedge^I$ is learnable in polynomial time with at most $|I|$ membership queries.*

Proof Consider the (dual) Algorithm SPEX in Figure 3 in Appendix A. Let F be the target function. Let $G_{\max} = f_1 \wedge f_2 \wedge \cdots \wedge f_t$ where $f_i \in \mathcal{F}^I$. By Lemma 26, we may assume w.l.o.g. that $G^{(i)} = f_1 \wedge f_2 \wedge \cdots \wedge f_{i-1} \wedge f_{i+1} \wedge \cdots f_t$ where $i = 1, \ldots, \ell$ are all the immediate descendants of G. Let $a^{(i)}$ be the witness for G and $G^{(i)}$, $i = 1, \ldots, \ell$.

In the algorithm, $S = \{f_i \mid i = 1, \ldots, t\}$. If $F(a^{(i)}) = 1$, then Line 5 in the algorithm removes f_i from S and f_i never returns to S. If $F(a^{(i)}) = 0$, then the set $\{f_1, f_2, \ldots, f_{i-1}, f_{i+1}, \cdots f_t\}$ is added to T, which means (see Line 3) that SPEX never considers a descendant that does not contain f_i. Namely, for every f_i, SPEX makes at most one membership query. ■

We conclude this section by illustrating SPEX on an example, depicted in Figure 7. Assume the set of boolean functions is $\mathcal{F}^I$, where $I = \{(1,2),(1,4),(1,3),(3,4),(2,4),(3,2)\}$, and the target is $G_{min} = 1$. The graph in Figure 7 shows the Hasse diagram (in white and gray nodes) and the candidates that SPEX considers (in gray). The figure demonstrates that the number of membership queries is equal to $|I|$.

6.2. Cyclic Sets

In this section, we consider the general case, where $I \subseteq [n]^2$ can be any set. Lemma 29 shows a few results when I is cyclic.

Lemma 29 *Let $I \subseteq [n]^2$ be any set with cycles. Then:*

1. *$G_{\max} = 0$ is in $\mathcal{F}_\wedge^I$.*
2. *The immediate descendants of $G_{\max}$ are all $\wedge_{(i,j)\in J}[x_i > x_j]$ where $\mathcal{G}_J$ is a maximal acyclic subgraph of $\mathcal{G}_I$.*
   In particular,
3. *Finding all the immediate descendants of $G_{\max}$ is equivalent to enumerating all the maximal acyclic subgraphs of $\mathcal{G}_I$.*

Proof If $i_1 \to i_2 \to \cdots \to i_c \to i_1$ is a cycle, then $G_{\max} \Rightarrow [x_{i_1} > x_{i_1}] = 0$ and thus $G_{\max} = 0$.

If $\mathcal{G}_J$ is a maximal acyclic subgraph of $\mathcal{G}_I$, then adding any edge in $I \backslash J$ to $\mathcal{G}_J$ creates a cycle. This implies that for any $[x_i > x_j] \in S(F_I) \backslash S(F_J)$, we have $F_J \wedge [x_i > x_j] = 0 = G_{\max}$. By Lemma 14, F_J is an immediate descendant of $G_{\max}$.

Now, if F_J is an immediate descendant of $G_{\max}$, then J is acyclic because otherwise $F_J = 0 = G_{\max}$. If $\mathcal{G}_J$ is not a maximal acyclic subgraph of $\mathcal{G}_I$, then there is an edge (i,j) such that $J \cup \{(i,j)\}$ is acyclic and then either $F_{J\cup\{(i,j)\}} = F_J$ – in which case F_J is not a representative and thus not an immediate descendant – or $F_{J\cup\{(i,j)\}} \neq F_J$ – in which case $G_{\max} \Rightarrow F_{J\cup\{(i,j)\}} \Rightarrow F_J$ and $G_{\max} \neq F_{J\cup\{(i,j)\}} \neq F_J$, and therefore, F_J is not an immediate descendant of $G_{\max}$. ■

Let $\mathcal{G}$ be any directed graph and denote by $N(\mathcal{G})$ the number of the maximal acyclic subgraphs of $\mathcal{G}$. Lemma 30 follows immediately from Theorem 12 and Lemma 29.

Lemma 30 $\mathrm{OPT}(\mathcal{F}_\wedge^I) \geq N(\mathcal{G}_I)$.

The problem of enumerating all the maximal acyclic subgraphs of a directed graph is still an open problem (Acua et al., 2012; Borassi et al., 2013; Wasa, 2016). We show that learning a function in $\mathcal{F}_\wedge^I$ (where $I \subseteq [n]^2$) in polynomial time is possible if and only if the enumeration problem can be done in polynomial time (the proof is in Appendix C).

Theorem 31 *There is a polynomial time learning algorithm (poly$(\mathrm{OPT}(\mathcal{F}_\wedge^I), n, |I|)$), which for an input $I \subseteq [n]^2$, learns $F \in \mathcal{F}_\wedge^I$ if and only if there is an algorithm that for an input $\mathcal{G}$, which is a directed graph, enumerates all the maximal acyclic subgraphs of $\mathcal{G}(V,E)$ in polynomial time (poly$(N(\mathcal{G}), |V|, |E|)$).*

7. Application to Program Synthesis

In this section, we explain the natural integration of SPEX into program synthesis. We then demonstrate this on a synthesizer that synthesizes programs that detect patterns in time-series charts. These programs meet specifications that belong to the class of variable inequalities I (for acyclic I).

Program synthesizers are defined over an input domain X_{in}, an output domain X_{out}, and a domain-specific language D. Given a specification, the goal of a synthesizer is to generate a corresponding program. A specification is a set of formulas $\varphi(x_{in}, x_{out})$ where x_{in} is interpreted over X_{in} and x_{out} is interpreted over X_{out}. Given a specification Y, a synthesizer returns a program $P : X_{in} \to X_{out}$ over D such that for all $in \in X_{in}$: $(in, P(in)) \models Y$ (i.e., all formulas are satisfied for $x_{in} = in$ and $x_{out} = out$). Roughly speaking, there are two types of synthesizers:

- Synthesizers that assume that Y describes a full specification. Namely, for all $in \in X_{in}$, there exists a single $out \in X_{out}$ such that $(in, out) \models Y$ (e.g., Solar-Lezama et al. (2008); Singh and Solar-Lezama (2011); Alur et al. (2013); Bornholt et al. (2016)).
- Synthesizers that assume that Y describes only input–output examples (known as PBE synthesizers). Namely, all formulas in the specification take the form of $x_{in} = in \Rightarrow x_{out} = out$ (e.g., Gulwani (2011); Polozov and Gulwani (2015); Barowy et al. (2015)). The typical setting of a PBE synthesizer is that an end user (that acts as the teacher) knows a target program f and he or she provides the synthesizer with some initial examples and can interact with the synthesizer through membership queries (we note that most synthesizers do not interact).

Each approach has its advantages and disadvantages. The first approach guarantees correctness on all inputs, but requires a full specification, which is complex to provide, especially by end users unfamiliar with formulas. On the other hand, PBE synthesizers are user-friendly as they interact through examples; however, generally they do not guarantee correctness on all inputs.

We next define the class of programs that are *$\mathcal{F}$-describable*. For such programs, SPEX can be leveraged by both approaches to eliminate their disadvantage. Let $\mathcal{F}$ be a set of predicates over $X_{in} \cup X_{out}$. A synthesizer is $\mathcal{F}$-describable if every program that can be synthesized meets a specification $F \in \mathcal{F}_\vee$ (or dually, $\mathcal{F}_\wedge$). A synthesizer that assumes that Y is a full specification and is $\mathcal{F}$-describable can release the user from having to provide the full specification by first running SPEX and then synthesizing a program from the formula returned by SPEX. A PBE synthesizer that is $\mathcal{F}$-describable can be extended to guarantee correctness on all inputs by first running SPEX and then synthesizing the program from the set of membership queries posed by SPEX. Theorem 32 follows immediately from Theorem 10.

Theorem 32 *Let $\mathcal{F}$ be a set of predicates and $\mathcal{A}$ be an $\mathcal{F}$-describable synthesizer. Then, $\mathcal{A}$ extended with SPEX returns the target program with at most $|\mathcal{F}| \cdot \max_{G \in G(\mathcal{F}_\vee)} |\mathrm{De}(G)|$ membership queries.*

7.1. Example: Synthesis of Time-series Patterns

In this section, we consider the setting of synthesizing programs that detect time-series patterns. The specifications of these programs are over $\mathcal{F}^I_\wedge$ for some $I \subseteq [n]^2$ and thus the synthesizer learns the target program within $|I|$ membership queries. Time-series are used in many domains including financial analysis (Bulkowski, 2005), medicine (Chuah and Fu, 2007), and seismology (Morales-Esteban et al., 2010). Experts use these charts to predict important events (e.g., trend changes in stock prices) by looking for *patterns* in the charts. There are a variety of platforms that enable users to write programs to detect a pattern in a time-series chart. In this work, we consider a domain-specific language (DSL) of a popular trading platform, AmiBroker. Our synthesizer can easily be extended to other DSLs.

A time-series chart $c : \mathbb{N} \to \mathbb{R}$ maps points in time to real values (e.g., stock prices). A time-series pattern is a conjunction F_I for acyclic I. The size of F_I is the maximal natural number it contains, i.e., the size of F_I is $argmax_i\{i \mid \exists j.(i, j) \in I$ or $(j, i) \in I\}$. A program detects a pattern F_I of size k in a time-series chart c if it alerts upon every $t \in \mathbb{N}$ for which the $t_1, ..., t_{k-1}$ preceding extreme points satisfy $F_I(c(t_1), ..., c(t_{k-1}), c(t)) = 1$.

In this setting, X_{in} is a set of charts over a fixed $k \in \mathbb{N}$, that is $f : \{1, ..., k\} \to \mathbb{R}$, and $X_{out} = \{0, 1\}$. The DSL D is the DSL of the trading platform AmiBroker. We built a synthesizer that not only interacts with the end user through membership queries, but also displays them as charts (of size k). Thereby, our synthesizer communicates with the end user in his language of expertise. The synthesizer takes as input an initial chart example $c' : \{1, ..., k\} \to \mathbb{R}$ and initializes I to $\{(i, j) \in [k]^2 \mid c'(i) \geq c'(j)\}$ and sets $\mathcal{F}^I := \{[x_i \geq x_j] \mid (i, j) \in I\}$ (our results are true also for these kinds of predicates). It then executes SPEX to learn F. During the execution, every witness is translated into a chart (the translation is immediate since each witness is an assignment to k points). Finally, our synthesizer synthesizes a program by synthesizing instructions that detect the k extreme points in the chart, followed by an instruction that checks whether these points satisfy the formula F and alerts the end user if so (the technical details are beyond the scope of this paper). Namely, for the end user, our synthesizer acts as a PBE synthesizer, but internally it takes the first synthesis approach and assumes it is given a full specification (which is obtained by running SPEX).

The complexity of the overall synthesis algorithm is determined by SPEX (as the synthesis merely synthesizes the instructions according to the specification F), and thus from Theorem 28, we infer the following theorem.

Theorem 33 *The pattern synthesizer returns a program that detects the target pattern in polynomial time with at most k^2 membership queries, where k is the target pattern size.*

8. Related Work

Program Synthesis Program synthesis has drawn a lot of attention over the last decade, especially in the setting of synthesis from examples, known as PBE (e.g., Gulwani (2010); Lau et al. (2003); Das Sarma et al. (2010); Harris and Gulwani (2011); Gulwani (2011); Gulwani et al. (2012); Singh and Gulwani (2012); Yessenov et al. (2013); Albarghouthi et al. (2013); Zhang and Sun (2013); Menon et al. (2013); Le and Gulwani (2014); Barowy et al. (2015); Polozov and Gulwani (2015)). Commonly, PBE algorithms synthesize programs consistent with the examples, which may not capture the user intent. Some works, however, guarantee to output the target program. For example, CEGIS (Solar-Lezama, 2008) learns a program via equivalence queries, and oracle-based synthesis (Jha et al., 2010) assumes that the input space is finite, which allows it to guarantee correctness by exploring all inputs (i.e., without validation queries). Synthesis has also been studied in a setting where a specification and the program's syntax are given and the goal is to find a program over this syntax meeting the specification (e.g., Solar-Lezama et al. (2008); Singh and Solar-Lezama (2011); Alur et al. (2013); Bornholt et al. (2016)).

Queries over Streams Several works aim to help analysts. Many trading software platforms provide domain-specific languages for writing queries where the user defines the query and the system is responsible for the sliding window mechanism, e.g., *MetaTrader*, *MetaStock*, *NinjaTrader*, and *Microsoft's StreamInsight* (Chandramouli et al., 2010). Another tool designed to help analysts is *Stat!* (Barnett et al., 2013), an interactive tool enabling analysts to write queries in StreamInsight. *TimeFork* (Badam et al., 2016) is an interactive tool that helps analysts with predictions based on automatic analysis of the past stock price. *CPL* (Anand et al., 2001) is a Haskell-based high-level language designed for chart pattern queries. Many other languages support queries for streams. *SASE* (Wu et al., 2006) is a system designed for RFID (radio frequency identification) streams that offers a user-friendly language and can handle large volumes of data. *Cayuga* (Brenna et al., 2007) is a system for detecting complex patterns in streams, whose language is based on Cayuga algebra. *SPL* (Hirzel et al., 2013) is IBM's stream processing language supporting pattern detections. *ActiveSheets* (Vaziri et al., 2014) is a platform that enables Microsoft Excel to process real-time streams from within spreadsheets.

9. Conclusion

In this paper, we have studied the learnability of disjunctions $\mathcal{F}_\vee$ (and conjunctions) over a set of boolean functions $\mathcal{F}$. We have shown an algorithm SPEX that asks at most $|\mathcal{F}| \cdot OPT(\mathcal{F}_\vee)$ membership queries. We further showed two classes that SPEX can learn in polynomial time. We then showed a practical application of SPEX that augments PBE synthesizers, giving them the ability to guarantee to output the target program as the end user intended. Lastly, we showed a

synthesizer that learns time-series patterns in polynomial time and outputs an executable program, while interacting with the end user through visual charts.

Acknowledgements The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7) under grant agreement n° 615688–ERC-COG-PRIME.

References

Hasan Abasi, Ali Z. Abdi, and Nader H. Bshouty. Learning boolean halfspaces with small weights from membership queries. In *Algorithmic Learning Theory: 25th International Conference, ALT '14*, 2014.

Elias Abboud, Nader Agha, Nader H. Bshouty, Nizar Radwan, and Fathi Saleh. Learning threshold functions with small weights using membership queries. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, COLT '99, pages 318–322, 1999.

Vicente Acua, Etienne Birmel, Ludovic Cottret, Pierluigi Crescenzi, Fabien Jourdan, Vincent Lacroix, Alberto Marchetti-Spaccamela, Andrea Marino, Paulo Vieira Milreu, Marie-France Sagot, and Leen Stougie. Telling stories: Enumerating maximal directed acyclic graphs with a constrained set of sources and targets. *Theoretical Computer Science*, 457:1 – 9, 2012.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV '13*, pages 934–950, 2013.

Noga Alon, Richard Beigel, Simon Kasif, Steven Rudich, and Benny Sudakov. Learning a hidden matching. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, 2002.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD '13*, pages 1–8, 2013.

AmiBroker. https://www.amibroker.com/.

Saswat Anand, Wei-Ngan Chin, and Siau-Cheng Khoo. Charting patterns on price history. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*, pages 134–145, 2001.

Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

Dana Angluin and Jiang Chen. Learning a hidden graph using queries per edge. *Journal of Computer and System Sciences*, 74(4):546 – 556, 2008. Carl Smith Memorial Issue.

Sriram Karthik Badam, Jieqiong Zhao, Shivalik Sen, Niklas Elmqvist, and David S. Ebert. Timefork: Interactive prediction of time series. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5409–5420, 2016.

Mike Barnett, Badrish Chandramouli, Robert DeLine, Steven Drucker, Danyel Fisher, Jonathan Goldstein, Patrick Morrison, and John Platt. Stat!: An interactive analytics environment for big data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD '13*, pages 1013–1016, 2013.

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, pages 218–228, 2015.

E. Biglieri and L. Gyrfi. *Multiple Access Channels: Theory and Practice*. IOS Press, 2007.

Annalisa De Bonis, Leszek Gasieniec, and Ugo Vaccaro. Optimal two-stage algorithms for group testing problems. volume 34, pages 1253–1270, 2005.

Michele Borassi, Pierluigi Crescenzi, Vincent Lacroix, Andrea Marino, Marie-France Sagot, and Paulo Vieira Milreu. Telling stories fast. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms: 12th International Symposium, SEA '13*, pages 200–211, 2013.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pages 775–788, 2016.

Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1100–1102, 2007.

Thomas N. Bulkowski. *Encyclopedia of Chart Patterns*. Wiley, 2nd edition, 2005.

Badrish Chandramouli, Jonathan Goldstein, and David Maier. High-performance dynamic pattern matching over disordered streams. In *PVLDB*, volume 3, pages 220–231, 2010.

Mooi Choo Chuah and Fen Fu. Ecg anomaly detection via time series analysis. In *Frontiers of High Performance Computing and Networking ISPA 2007 Workshops: ISPA 2007 International Workshops SSDSN, UPWN, WISH, SGC, ParDMCom, HiPCoMB, and IST-AWSN*, pages 123–135, 2007.

Ferdinando Cicalese. Group testing. In *Fault-Tolerant Search Algorithms*, pages 139–173. Springer, 2013.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In *Database Theory - ICDT '10, 13th International Conference*, pages 89–103, 2010.

Robert Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.

D. Du and F. Hwang. *Combinatorial Group Testing and Its Applications*. Applied Mathematics. World Scientific, 2000.

D. Du and F. Hwang. *Pooling Designs and Nonadaptive Group Testing: Important Tools for DNA Sequencing*. Series on applied mathematics. World Scientific, 2006.

Vladimir Grebinski and Gregory Kucherov. Reconstructing a hamiltonian cycle by querying the graph: Application to DNA physical mapping. *Discrete Appl. Math.*, 88(1-3):147–165, November 1998.

Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 13–24, 2010.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11*, pages 317–330, 2011.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pages 317–328, 2011.

Tibor Hegedűs. Generalized teaching dimensions and the query complexity of learning. In *Proceedings of the Eighth Annual Conference on Computational Learning Theory*, COLT '95, pages 108–117, 1995.

M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM streams processing language: Analyzing big data in motion. *IBM J. Res. Dev.*, 57(3-4), 2013.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 215–224, 2010.

Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

Tessa A. Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 542–553, 2014.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML '13*, pages 187–195, 2013.

A. Morales-Esteban, F. Martnez-lvarez, A. Troncoso, J.L. Justo, and C. Rubio-Escudero. Pattern recognition to forecast seismic time series. *Expert Systems with Applications*, 37(12):8333 – 8342, 2010.

Hung Q Ngo and Ding-Zhu Du. A survey on combinatorial group testing algorithms with applications to DNA library screening. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 2000.

Andrzej Pelc. Searching games with errors—fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, January 2002.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '15*, pages 107–126, 2015.

Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751, 2012.

Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*, pages 289–299, 2011.

Armando Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 136–148, 2008.

Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference*, pages 360–384. 2014.

Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, abs/1605.05102, 2016.

Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 407–418, 2006.

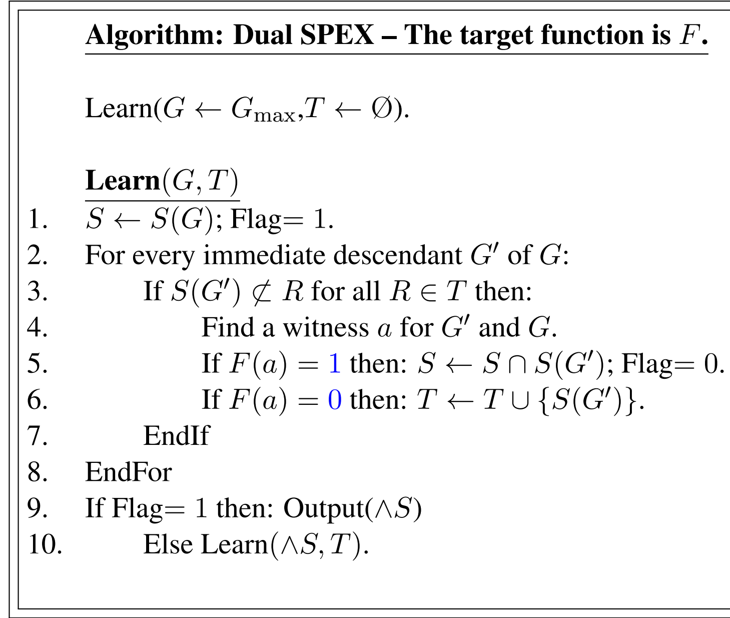
Kuat Yessenov, Shubham Tulsiani, Aditya Krishna Menon, Robert C. Miller, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A colorful approach to text processing by example. In *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13*, pages 495–504, 2013.

Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE '13*, pages 224–234, 2013.

N. Yu. Zolotykh and V. N. Shevchenko. Deciphering threshold functions of k-valued logic. In *Discrete Analysis and Operations Research. Novosibirsk 2(3), pp. 18. English translation: Korshunov, A. D. (ed.): Operations Research and Discrete Analysis. Kluwer Ac. Publ. Netherlands. (1997)*, 1995.

Appendix A. The Dual SPEX Algorithm

Figure 3 shows the dual SPEX algorithm for learning functions in $\mathcal{F}_\wedge$.



Algorithm: Dual SPEX – The target function is F.

Learn($G \leftarrow G_{\max}$,$T \leftarrow \emptyset$).

Learn(G,T)

1. $S \leftarrow S(G)$; Flag$=1$.
2. For every immediate descendant G' of G:
3. If $S(G') \not\subset R$ for all $R \in T$ then:
4. Find a witness a for G' and G.
5. If $F(a)=1$ then: $S \leftarrow S \cap S(G')$; Flag$=0$.
6. If $F(a)=0$ then: $T \leftarrow T \cup \{S(G')\}$.
7. EndIf
8. EndFor
9. If Flag$=1$ then: Output($\wedge S$)
10. Else Learn($\wedge S, T$).

Figure 3: The dual algorithm of SPEX for learning functions in $\mathcal{F}_\wedge$.

Appendix B. Proofs for Section 3.2

Proof of Lemma 2 Consider G_F. Since $F=G_F$, $G_1 \Rightarrow G_F \Rightarrow G_2$. By the definition of immediate descendants, we get the result.
Proof of Lemma 3 Bullet *1*: Consider G_F. Then $G_F=F$ and $G_F \in G(\mathcal{F}_\vee)$. Since $G_1, G_2 \Rightarrow G_F \Rightarrow \mathrm{lca}(G_1,G_2)$, by the definition of LCA we must have $G_F=\mathrm{lca}(G_1,G_2)$. The proof of *2* is similar.
Proof of Lemma 4 Since $G_1, G_2 \Rightarrow \mathrm{lca}(G_1,G_2)$, we get $G_1 \vee G_2 \Rightarrow \mathrm{lca}(G_1,G_2)$. Since $G_1, G_2 \Rightarrow (G_1 \vee G_2) \Rightarrow \mathrm{lca}(G_1,G_2)$, by Lemma 3, we get $G_1 \vee G_2 = \mathrm{lca}(G_1,G_2)$.
Proof of Lemma 6 Let $G=\gcd(G_1,G_2)$. We show that $S(G) \subseteq S(G_1) \cap S(G_2)$ and $S(G_1) \cap S(G_2) \subseteq S(G)$. By Lemma 5, $S(G) \subseteq S(G_1)$ and $S(G) \subseteq S(G_2)$, and therefore, $S(G) \subseteq S(G_1) \cap S(G_2)$. Since $S(G) \subseteq S(G_1) \cap S(G_2)$, we also have $G=\vee S(G) \Rightarrow \vee(S(G_1) \cap S(G_2)) \Rightarrow G_1, G_2$. Therefore, by Lemma 3, we get $G=\vee(S(G_1) \cap S(G_2))$. Thus, $S(G_1) \cap S(G_2) \subseteq S(G)$.

Appendix C. Additional Proofs for Section 6

Proof of Lemma 24 *Only if:* Assume $F_1 = F_2$. Suppose, on the contrary, that there are i, j such that w.l.o.g. $R(\mathcal{I}(F_1))_{i,j} = 0$ and $R(\mathcal{I}(F_2))_{i,j} = 1$. Since I is acyclic and $R(\mathcal{I}(F_2))_{i,j} = 1$, there is no path from j to i in $\mathcal{G}_I$ (and therefore, in $\mathcal{G}_{\mathcal{I}(F_1)}$). Since $R(\mathcal{I}(F_1))_{i,j} = 0$, there is also no path from i to j in $\mathcal{G}_{\mathcal{I}(F_1)}$. Therefore, we can match the vertices i and j in $\mathcal{G}_{\mathcal{I}(F_1)}$ (unify them into a single vertex) and get an acyclic graph $\mathcal{G}'$. Using the topological sorting of $\mathcal{G}'$, we get a satisfying assignment a for F_1 that satisfies $a_i = a_j$. We now show that $F_2(a) = 0$ and thus get a contradiction. Since $R(\mathcal{I}(F_2))_{i,j} = 1$, there is a path $i = i_1 \to i_2 \to \cdots \to i_\ell = j$ from i to j in $\mathcal{G}_{\mathcal{I}(F_2)}$. Therefore, F_2 contains $F' := [x_{i_1} > x_{i_2}] \wedge [x_{i_2} > x_{i_3}] \wedge \cdots \wedge [x_{i_{\ell-1}} > x_{i_\ell}]$. Since $F_2 \Rightarrow F' \Rightarrow [x_{i_1} > x_{i_\ell}] = [x_i > x_j]$ and our assignment satisfies $[a_i > a_j] = 0$, we get $F_2(a) = 0$.

If: Assume $R(\mathcal{I}(F_2)) = R(\mathcal{I}(F_1))$. Suppose, on the contrary, that $F_2 \neq F_1$. Then, there is an assignment a such that $F_2(a) = 1$ and $F_1(a) = 0$ (or vice versa). Since $F_1(a) = 0$, a is not a topological sorting of $\mathcal{G}_{\mathcal{I}(F_1)}$. Therefore, there is an edge $i \to j$ in $\mathcal{G}_{\mathcal{I}(F_1)}$ such that $a_i \le a_j$. Since $R(\mathcal{I}(F_2))_{i,j} = R(\mathcal{I}(F_1))_{i,j} = 1$, there is a path from i to j in $\mathcal{G}_{\mathcal{I}(F_2)}$. As before, we get a contradiction.

Proof of Lemma 26 Since $(r, s) \in \mathcal{I}(G)$, we have $R(\mathcal{I}(G))_{r,s} = 1$. On the other hand, since there is no path from r to s in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(r,s)\}}$, we have $R(\mathcal{I}(G^{r,s}))_{r,s} = 0$. Therefore, $R(\mathcal{I}(G)) \neq R(\mathcal{I}(G^{r,s}))$ and by Lemma 24, we get $G \neq G^{r,s}$. By Lemma 14, $G^{r,s}$ is an immediate descendant of G.

To show that there is no other immediate descendant, we use (the dual result of) Lemma 16. Note that $S(G)\backslash S(G^{r,s}) = \{[x_r > x_s]\}$ and thus, by Lemma 16, it is sufficient to prove that $G = G' := \wedge_{(i,j)\in J}[x_i > x_j]$, where $J = \{(i,j) \in \mathcal{I}(G) \mid$ there is no path from i to j in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(i,j)\}}\}$. To prove it, we show $R(\mathcal{I}(G')) = R(J)$, and then the result follows from Lemma 24.

If $R(J)_{i,j} = 1$, then $R(\mathcal{I}(G'))_{i,j} = 1$ since $\mathcal{G}_{\mathcal{I}(G')}$ is a subgraph of $\mathcal{G}_{\mathcal{I}(G)}$. If $R(\mathcal{I}(G'))_{i,j} = 1$, then there is a path from i to j in $\mathcal{G}_{\mathcal{I}(G')}$, and therefore, there is a path from i to j in $\mathcal{G}_{\mathcal{I}(G)}$, and thus $R(\mathcal{I}(G))_{i,j} = 1$. Since $R(\mathcal{I}(G))_{i,j} = 1$, there is a path p from i to j in $\mathcal{G}_{\mathcal{I}(G)}$. Let $(r, s) \notin \mathcal{I}(G)\backslash J$. Then, $(r, s) \in \mathcal{I}(G)$ and there is a path (other than $r \to s$) $r \to v_1 \to v_2 \to \cdots \to v_\ell = s$ in $\mathcal{G}_{\mathcal{I}(G)}$. We now show that there is a path from i to j in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(r,s)\}}$. This is true because if the path p (in $\mathcal{G}_{\mathcal{I}(G)}$) contains the edge $r \to s$, then we can replace this edge with the path $r \to v_1 \to v_2 \to \cdots \to v_\ell = s$ and get a new path from i to j in $\mathcal{G}_{\mathcal{I}(G)\backslash\{(r,s)\}}$. Therefore, $R(\mathcal{I}(G)\backslash\{(r,s)\})_{i,j} = 1$. By repeating this on the other edges in $\mathcal{I}(G)\backslash J$, we get $R(J)_{i,j} = 1$.

Proof of Theorem 31: *If:* Let $\mathcal{A}$ be an algorithm that for an input $\mathcal{G}$, which is a directed graph, enumerates all the maximal acyclic subgraphs in polynomial time (poly$(N(\mathcal{G}), |V|, |E|)$). The first step of SPEX (in Figure 3) finds all the immediate descendants of $G_{\max}$. By Lemma 29, this is equivalent to enumerating all the maximal acyclic subgraphs of $\mathcal{G}_I$. This can be done by $\mathcal{A}$ in time *poly*$(N(\mathcal{G}_I), n, |I|)$. For every immediate descendant G' of $G_{\max} = 0$, any topological sorting of G' is a witness for G' and G. Once SPEX calls Learn on one of the immediate descendants of $G_{\max}$, the algorithm proceeds as in the acyclic case. This algorithm runs in time *poly*$(N(\mathcal{G}_I), n, |I|)$ time and asks at most $N(G_I) + |I|$ membership queries. By Lemma 30, the algorithm runs in time *poly*$(\mathrm{OPT}(\mathcal{F}_\wedge^I), n, |I|)$ and asks at most $\mathrm{OPT}(\mathcal{F}_\wedge^I) + |I|$ queries.

Only if: Let $\mathcal{B}$ be a learning algorithm that runs in poly$(\mathrm{OPT}(\mathcal{F}_\wedge^I), n, |I|)$. By the above argument:

$$\mathrm{OPT}(\mathcal{F}_\wedge^I) \le N(\mathcal{G}_I) + |I|. \tag{3}$$

Let $\mathcal{G} = ([n], E)$ be any directed graph. We run the learning algorithm with the target F_I where $I = E$. For any membership query asked by the algorithm, we answer 0 until the algorithm outputs the hypothesis $G_{\max} = 0$. Suppose A is the set of all membership queries that are asked by the algorithm. We now claim that:

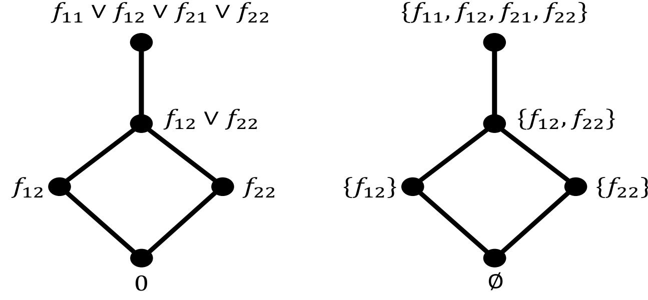


Figure 4: Left: the Hasse diagram of Ray_2^2. Right: the corresponding $S(G)$ sets.

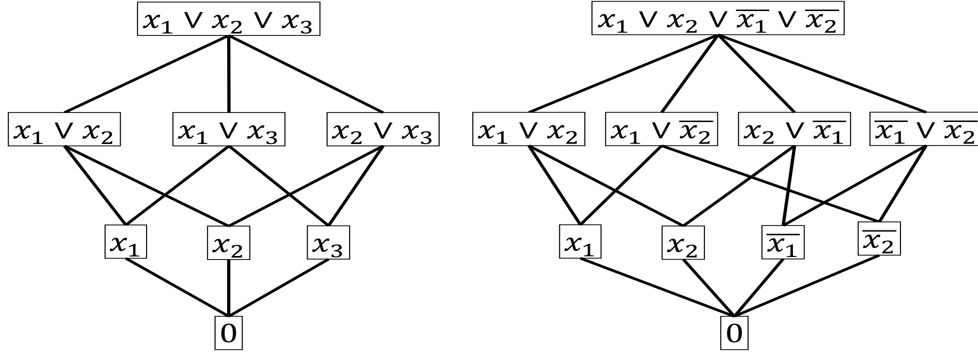


Figure 5: Hasse diagram of terms and monotone terms.

1. $|A| = poly(N(\mathcal{G}), n, |E|)$.
2. If G' is a maximal acyclic subgraph of G, then there is an assignment $a \in A$ such that $E(G') = \{(i, j) \in E \mid a_i > a_j\}$, where $E(G')$ is the set of edges of G'.

Bullet 1 follows since $\mathcal{B}$ runs in time $poly(\mathrm{OPT}(\mathcal{F}_\wedge^I), n, |I|)$ and by (3) this is $poly(N(\mathcal{G}), n, |E|)$. So the number of membership queries cannot be more than $poly(N(\mathcal{G}), n, |E|)$ time.

We now prove bullet 2. There is an assignment $a \in A$ that satisfies $F_{E(G')}(a) = 1$, because otherwise the algorithm cannot distinguish between $F_{E(G')}$ and $G_{\max}$, which violates the correctness of the algorithm. Now, since $F_{E(G')}(a) = 1$, we must have $E(G') \subseteq \{(i, j) \in E \mid a_i > a_j\}$. Since $E(G')$ is maximal (adding another edge will create a cycle), we get $E(G') = \{(i, j) \in E \mid a_i > a_j\}$.

The algorithm that enumerates all the maximal acyclic subgraphs of $\mathcal{G}(V, E)$ continues to run as follows: for each $a \in A$, it defines $E_a := \{(i, j) \in E \mid a_i > a_j\}$. If $G_a := ([n], E_a)$ is a maximal cyclic subgraph, then it lists G_a. Testing whether $G_a := ([n], E_a)$ is maximal can be done in polynomial time (e.g., by checking edge-by-edge in E). It is easy to verify that the algorithm runs in poly$(N(\mathcal{G}), |V|, |E|)$ time.

Appendix D. Additional Figures

Here, we provide Figures 4, 5, 6, and 7.

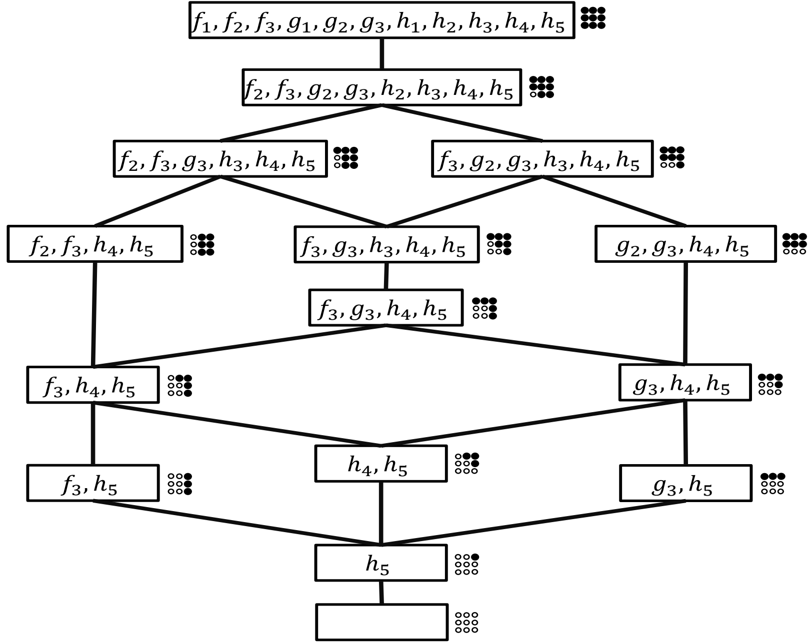


Figure 6: Hasse diagram of $\mathcal{F}=\{f_1, f_2, f_3, g_1, g_2, g_3, h_1, \ldots, h_5\}$ whose functions are $\{1,2,3\}\times\{1,2,3\}\to\{0,1\}$ where $f_i(x_1, y_1)=[x_1 \geq i]$, $g_i(x_1, x_2)=[x_2 \geq i]$ and $h_i(x_1, x_2)=[x_1 + x_2 \geq i+1]$.

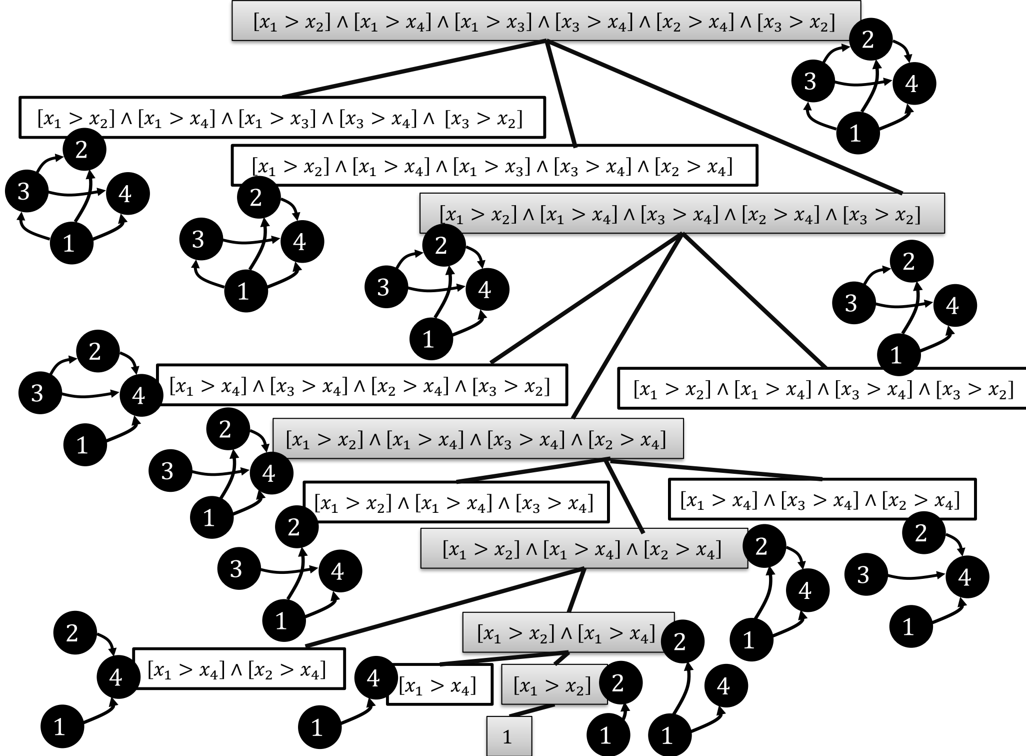


Figure 7: A path from $G_{\max}$ to $G_{\min}$ in the inequality predicate diagram. Here $\mathcal{I}(G_{\max}) = \{(1,2),(1,4),(1,3),(3,4),(2,4),(3,2)\}$