# Logic Programs with Abstract Constraint Atoms: The Role of Computations

Lengning Liu[1], Enrico Pontelli[2], Tran Cao Son[2], and Mirosław Truszczyński[1]

[1] Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA, {lliu1,mirek}@cs.uky.edu
[2] Department of Computer Science, New Mexico State University, Las Cruces, NM 88003, USA, {epontell,tson}@cs.nmsu.edu

**Abstract.** We provide new perspectives on the semantics of logic programs with *constraints*. To this end we introduce several notions of *computation* and propose to use the *results* of computations as answer sets of programs with constraints. We discuss the rationale behind different classes of computations and study the relationships among them and among the corresponding concepts of answer sets. The proposed semantics generalize the answer set semantics for programs with monotone, convex and/or arbitrary constraints described in the literature.

## 1 Introduction and Motivation

In this paper we study logic programs with *arbitrary* abstract constraints (which we also simply refer to as *constraints*). Programs with constraints generalize normal logic programs, programs with monotone and convex constraints [11, 16], and several classes of programs with aggregates (e.g., [2, 6, 18]). Intuitively, a constraint $A$ represents a *condition* on models of the program containing $A$. The definition of $A$ includes an *explicit* description of conditions interpretations have to meet in order to satisfy it. The syntax of such programs and one possible semantics have been proposed in [15]. Another semantics has been proposed in [21]. Following the approach proposed in [11], and exploiting analogies to the case of normal logic programs, we introduce several other semantics based on *computations* for programs with constraints. We argue that the *results* of (some types of) computations adequately generalize *answer sets* of programs with constraints.

The notion of an *answer set* of a logic program [8] is the foundation for answer-set programming (ASP) [14, 17]. Intuitively, an answer set represents beliefs of an agent, given a logic program encoding its knowledge base. Researchers developed several characterizations of answer sets, providing a reasoner with alternative ways to determine them. The original definition of answer sets [8] naturally leads to a "guess-and-check" approach. We first guess a candidate for an answer set, and then we validate the guess. The validation consists of recomputing the guess starting with the empty set and iterating the *one-step provability operator* [22] for the *Gelfond-Lifschitz* program reduct [8]. Alternatively, we can compute an answer set starting with the empty set. At each step, we include in the set under construction the heads of *some* of the rules applicable at this step; typically, we include all the rules selected during the previous steps *plus* some additional ones. Once the process stabilizes, we need to check that the final result does not block any rules chosen to fire earlier [12, 13]. In this second approach, we replace the initial non-deterministic step of guessing the answer set with non-deterministic choices of rules to fire at each step of the construction.

*Example 1.* Let us consider the program $P_1$ consisting of the following rules:

$$a \leftarrow \mathbf{not}\ b \qquad c \leftarrow a \qquad b \leftarrow \mathbf{not}\ a \qquad d \leftarrow b.$$

This program has two answer sets: $\{a, c\}$ and $\{b, d\}$. In the "guess-and-check" approach, we might guess $\{a, c\}$ as a candidate answer set. To verify the guess, we compute the Gelfond-Lifschitz reduct, consisting of the rules $a \leftarrow$ , $c \leftarrow a$ and $d \leftarrow b$. Next, we iterate the one-step provability operator for the reduct to compute its least Herbrand model—which corresponds to $\{a, c\}$. Since it coincides with the initial guess, the guess is validated as an answer set. In this way, we can also validate the guess $\{b, d\}$. However, the validation of $\{a\}$ fails—i.e., $\{a\}$ is not an answer set.

The other approach starts with the empty interpretation, $\emptyset$, which makes two rules applicable: $a \leftarrow \mathbf{not}\ b$ and $b \leftarrow \mathbf{not}\ a$. The algorithm needs to select some of them to "fire", say, it selects $a \leftarrow \mathbf{not}\ b$. The choice results in the new interpretation, $\{a\}$. Two rules are applicable now: $a \leftarrow \mathbf{not}\ b$ and $c \leftarrow a$. The algorithm selects both rules for firing. The interpretation that results is $\{a, c\}$. The same two rules that were applicable in the previous step are applicable now. Thus, there is no possibility to add new elements to the current set. The computation stabilizes at $\{a, c\}$. Since $\{a, c\}$ does not block any of the rules fired in the process, $\{a, c\}$ is an answer set. $\square$

We note that the first approach starts with a tentative answer set of the program, while the second starts with the *empty* interpretation. In the first approach, we guess the entire answer set at once and, from that point on, proceed in a deterministic fashion. In the second approach we construct an answer set *incrementally* making non-deterministic choices along the way. Thus, each approach involves non-determinism. However, in the second approach, the role of non-determinism is generally reduced. In this paper we cast these two approaches in terms of abstract principles related to a notion of *computation*. We then lift these principles to the case of programs with abstract constraints and propose several new semantics for such programs.

The interest in ASP has been fueled by the development of software to compute answer-sets of logic programs, most notably SMODELS and DLV, which allow programmers to tackle complex real-world problems (e.g., [1, 5, 10]). To facilitate declarative solutions of problems in knowledge representation and reasoning, researchers proposed extensions of the logic programming language, which support constraints and aggregates [2–4, 6, 9, 17–19]. This development stimulated interest in logic programming formalisms based on *abstract constraint atoms* [15, 16]. The concept of an abstract constraint atom is general and encompasses several language-level extensions including aggregates. The introduction of constraints brought forth the question of how to extend the semantics of answer sets to the case of programs with constraints. Researchers proposed several possible approaches [3, 4, 6, 19–21]. They *all* agree on specific classes of programs with constraints including normal logic programs (viewed as programs with constraints), programs with *monotone* constraints [16], and programs with *convex* constraints [11]. However, they differ on programs with arbitrary constraints.

What makes the task of defining answer sets for programs with arbitrary constraints difficult and interesting is the *nonmonotonic* behavior of such constraints. For instance, let us consider the constraint $(\{p(1), p(-1)\}, \{\{p(1)\}\})$ (we introduce this notation in Section 3), which can be seen as an encoding of the aggregate $\textsc{Sum}(\{X \mid p(X)\}) \geq 1$.[3] This aggregate atom is true in the interpretation $\{p(1)\}$ but false in $\{p(1), p(-1)\}$.

The contribution of this paper is the development of a general framework for defining and studying answer sets for logic programs with arbitrary constraints. Our proposals rely on an abstract notion of incremental *computation* and can be traced back to one of the two basic approaches to computing answer sets of normal logic programs that we mentioned above. This notion generalizes an approach developed in [11, 16] for the case of programs with monotone and convex constraints. In the paper, we study properties of the notions of answer set we introduce, and relate them to the earlier proposals.

## 2 Computations in Normal Logic Programs—Principles

We start by motivating the notion of a *computation*, which is central to our paper. To this end, we look at the case of normal logic programs and build on our discussion in Example 1. In particular, we show how to use computations to characterize answer-sets. We represent propositional interpretations as sets of atoms. A program rule whose body is satisfied by an interpretation $M$ is called *M-applicable*. We write $P(M)$ to denote the set of all $M$-applicable rules in a program $P$. The *one-step provability operator* assigns to an interpretation $M$ the set of the heads of all rules in $P(M)$. We denote this operator by $T_P$. Fixpoints of $T_P$ are *supported* models of $P$. An interpretation $M$ is a *stable model* or, as we will say here, an *answer set* of $P$ if $M$ is the least model of the Gelfond-Lifschitz reduct $P^M$.

We define computations as sequences $\langle X_i \rangle_{i=0}^{\infty}$ of sets of atoms (propositional interpretations), where $X_i$ represents the status of the computation at step $i$. In particular, we *require* that $X_0 = \emptyset$. The basic intuition is that a computation, at each step $i \geq 1$, revises its previous status $X_{i-1}$. We base the revision on a non-deterministic operator, $Concl_P(X)$, that provides the set of revisions of $X$ that can be justified according to a logic program $P$ and a set of atoms $X$. Formally, a set of atoms $Y$ is *grounded* in a set of atoms $X$ and a program $P$ if

$$Y \subseteq \{a \mid (a \leftarrow body) \in P \textit{ and } X \models body\}.$$

We write $Concl_P(X)$ to denote the set of all possible sets $Y$ grounded in $X$ and $P$. We require that computations satisfy the *principle of revision*:

> $(\mathbf{R})$ *Revision principle*: each successive element in a computation must be grounded in the preceding one and the program, i.e., $X_i \in Concl_P(X_{i-1})$, for every $i$, $1 \leq i$.

A computation of an answer set of a program, using a method as described in Example 1, produces a monotonically increasing sequence of sets, each being a part of the answer set being built. Thus, at each step not only new atoms are computed, but also all atoms established earlier are recomputed. This suggests the principle of *persistence of beliefs*:

> $(\mathbf{P})$ *Persistence of beliefs*: each next element in the computation must contain the previous one (once we "revise an atom in", we keep it), i.e., $X_{i-1} \subseteq X_i$, for every $i$, $1 \leq i$.

[3] We assume that $1, -1$ are the two available constants.

For a sequence $\langle X_i \rangle_{i=0}^{\infty}$ satisfying the principle $(\mathbf{P})$, we define $X_\infty$ to be the *result* of $\langle X_i \rangle_{i=0}^{\infty}$ by setting $X_\infty = \bigcup_{i=0}^{\infty} X_i$. The result of the computation should be an interpretation that could not be revised further. This suggests one more basic principle for computations, the principle of *convergence*:

> $(\mathbf{C})$ *Convergence*: the computation process continues until it becomes stable (no additional revisions can be made), i.e., $X_\infty = T_P(X_\infty)$, where $T_P$ is the one-step provability operator. In other words, $X_\infty$ is a supported model of $P$.

**Definition 1.** *Let $P$ be a normal logic program. A sequence $\langle X_i \rangle_{i=0}^{\infty}$ is a* computation *for $P$ if $\langle X_i \rangle_{i=0}^{\infty}$ satisfies the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$, and $X_0 = \emptyset$.*

Computations are indeed relevant to the task of describing answer sets of normal logic programs. We have the following result.

**Proposition 1.** *Let $P$ be a normal logic program. If a set of atoms $X$ is an answer set of $P$ then there exists a computation for $P$, $\langle X_i \rangle_{i=0}^{\infty}$, such that $X = X_\infty$.*

*Proof (Sketch).* The sequence $\langle X \rangle_{i=0}^{\infty}$ can be obtained from the iterations of the one-step provability operator of the Gelfond-Lifschitz reduct of the program $P$. □

Proposition 1 implies that the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ give a notion of computation broad enough to derive any answer set. Is this concept of computation what is needed to precisely characterize answer sets? In other words, does *every* sequence of sets of atoms starting with the $\emptyset$ and satisfying the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ result in an answer set? This is indeed the case for *positive* normal logic programs, more commonly referred to as Horn programs.

**Proposition 2.** *Let $P$ be a positive logic program. The result of every computation is equal to the least model of $P$, that is, to the unique answer set of $P$.*

However, in the case of arbitrary normal programs, there are computations that do not result in answer sets.

*Example 2.* Consider the program $P_2$ containing the two rules $a \leftarrow \mathbf{not}\ a$ and $a \leftarrow a$. The sequence $X_0 = \emptyset$, $X_1 = \{a\}$, $X_2 = \{a\}, \ldots$ satisfies $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ and so, it is a computation for $P$. However, $X = \bigcup_{i=0}^{\infty} X_i = \{a\}$ is not an answer set of $P_2$. □

It follows that the notion of a computation defined by the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ is too broad to describe exactly the notion of an answer set. Let us come back to Example 2. There, $a \in X_1$ because the body of the first rule is satisfied by the set $\emptyset$. However, the body of the first rule is *not* satisfied in every set $X_i$ for $i \geq 1$. Nevertheless, $a \in X_i$, for $i \geq 2$, since the body of the *second* rule is satisfied by $X_{i-1}$. Thus, the reason for the presence of $a$ in the next revision changes between the first and second step. This is why that sequence does not result in an answer set, even though it satisfies the principle $(\mathbf{P})$, which guarantees that atoms once revised in will remain in throughout the rest of the computation.

These considerations suggest that useful classes of computations can be obtained by requiring that not only atoms but also *reasons* for including atoms persist. Intuitively, we would like to associate with each atom included in $X_i$ a rule that supports the inclusion, and this rule should remain applicable from that point on. More formally, we state this principle as follows:

(**Pr**) *Persistence of Reasons*: for every $a \in X_\infty$ there is a rule $r_a \in P$ (called the *reason* for $a$) whose head is $a$ and whose body holds in every $X_i$, $i \geq i_a - 1$, where $i_a$ is the least integer such that $a \in X_{i_a}$.

This principle is exactly what is needed to characterize answer sets of normal logic programs.

**Definition 2.** *Let $P$ be a normal logic program. A computation $\langle X \rangle_{i=0}^{\infty}$ for $P$ is* persistent *if it satisfies the principle* $(\mathbf{Pr})$.

**Proposition 3.** *Let $P$ be a normal logic program. A set $X$ is an answer set of $P$ if and only if there is a persistent computation for $P$ such that its result is $X$.*

We now observe that, in general, the operator $Concl_P$ offers several choices for revising a current interpretation $X_{i-1}$ into $X_i$ during a computation. The question is whether this freedom is necessary or whether we can restrict the principle $(\mathbf{R})$ and still characterize answer sets of normal logic programs.

*Example 3.* Let $P_3$ be the normal logic program:

$$a \leftarrow \mathbf{not}\ b \qquad c \leftarrow \mathbf{not}\ b \qquad e \leftarrow a, c \qquad f \leftarrow a, \mathbf{not}\ c.$$

This program has only one answer set $M = \{a, c, e\}$, which corresponds to the computation $\emptyset$, $\{a, c\}$, $\{a, c, e\}$. In this computation, at each step $i = 1, 2$, we take as $X_i$ a greatest element of $Concl_{P_3}(X_{i-1})$, which exists and is given by $T_{P_3}(X_{i-1})$. Thus, the next element of the computation is the result of firing *all applicable* rules.

On the other hand, selecting an element in $Concl_{P_3}(X)$ other than $T_{P_3}(X)$ can result in sequences that cannot be extended to a computation. For example, the sequence $\emptyset$, $\{a\}$, $\{a, f\}$ represents a potential computation since it satisfies the (**R**) and (**P**) principles. Yet, no possible extension of this sequence satisfies the (**C**) principle. □

This example indicates that interesting classes of computations can be obtained by restricting the operator $Concl_P$. Since for every $X$ we have $T_P(X) \in Concl_P(X)$, we could restrict the choice for possible revisions of $X$ based on $P$ to $T_P(X)$. The class of computations obtained under this restriction is a *proper* subset of the class of computations. For instance, the program $P_1$ from Example 1 does not admit computations that revise $X_{i-1}$ into $X_i = T_P(X_{i-1})$. Thus, the class of such computations is not adequate for the task of characterizing answer sets of normal logic program. We note, though, that they do characterize answer sets for some special classes of logic programs, for instance, for stratified logic programs.

To obtain a general characterization of answer sets by restricting the choices offered by $Concl_P(X)$, we need to modify the operator $T_P(X)$. The first approach to computing answer sets, discussed in the introduction provides a clue: we need to change the notion of satisfiability used in the definition of $T_P(X)$.

Let $M \subseteq At$. We define the satisfiability relation $\models_M$, between sets of atoms and conjunctions of literals, as follows: we say that $S \models_M F$ holds (where $S \subseteq At$ and $F$ is a conjunction of literals) if $S \models F$ and $M \models F$. That is, the satisfaction is based not only on $S$ but also on $M$ (the "context"). We now define the (context-based) one-step provability operator $T_P^M$ as follows:

$$T_P^M(X) = \{a \mid a \leftarrow \mathit{body} \in P,\ X \models_M \mathit{body}\}.$$

We note that $T_P^M(X) \in \mathit{Concl}_P(X)$. Thus, we obtain the following result.

**Proposition 4.** *Let $P$ be a normal logic program. A sequence $\langle X_i \rangle_{i=0}^{\infty}$ of sets of atoms that satisfies properties $(\mathbf{P})$ and $(\mathbf{C})$, as well as the property $X_i = T_P^M(X_{i-1})$, for $i = 1, 2, \ldots$, is a computation for $P$.*

If a computation is determined by the satisfiability relation $\models_M$ in the way described in Proposition 4, we call it an $M$-computation. Not all $M$-computations define answer sets. Let us consider the program $P_2$ from Example 2. The sequence $X_0 = \emptyset$, $X_i = \{a\}$, $i = 1, 2, \ldots$, is a $\emptyset$-computation for $P_2$. But the result of this computation ($\{a\}$) is not an answer set for $P_2$.

The problem is that $M$-computations may not to be persistent. In fact, the $\emptyset$-computation described above is not. It turns out that if we impose the condition of persistence on $M$-computations, their results are answer sets.

**Proposition 5.** *Let $P$ be a normal logic program and $M \subseteq At$. If an $M$-computation is persistent then its result is an answer set for $P$.*

It is also the case that every answer set is the result of some persistent $M$-computation.

**Proposition 6.** *Let $P$ be a normal logic program. A set $M$ of atoms is an answer set of $P$ if and only if there exists a persistent $N$-computation whose result is $M$.*

A even stronger result can be proved—in which answer sets are characterized by a proper subclass of persistent $M$-computations. We call an $M$-computation *self-justified* if its result is $M$.

**Proposition 7.** *Let $P$ be a normal logic program. A set $M \subseteq At$ is an answer set of $P$ if and only if $P$ has a self-justifying $M$-computation (whose result, by the definition, is $M$).*

One can check that self-justified $M$-computations are persistent. Moreover, in general, the class of self-justified $M$-computations is a proper subclass of $M$-computations. Thus, we can summarize our discussion in this section as follows. Answer sets of normal logic programs can be characterized as the results of persistent computations, persistent $M$-computations, and self-justified $M$-computations, with each subsequent class being a proper subclass of the preceding one.

Our goal, in this section, was to recast answer sets of normal logic programs in terms of computations. More specifically, taking two ways of computing answer sets as the departure point, we introduced three characterizations of answer sets in terms of persistent computations. In Sections 4 and 5, we will show how to generalize the classes of computations discussed here to the case of programs with constraints. In this way, we will arrive at concepts of answer sets for programs with constraints that generalize the concept of answer sets for normal logic programs.

## 3 Programs with Abstract Constraints—Basic Definitions

We recall here some basic definitions concerning programs with constraints [11, 15, 16]. We fix an infinite set $At$ of propositional variables. A *constraint* is an expression $A = (X, C)$, where $X \subseteq At$ is a *finite* set, and $C \subseteq \mathcal{P}(X)$ ($\mathcal{P}(X)$ denotes the powerset of $X$). The set $X$ is called the *domain* of $A = (X, C)$, denoted by $A_{dom}$. Elements of $C$ are called *satisfiers* of $A$, denoted by $A_{sat}$. Intuitively, the sets in $A_{sat}$ are precisely those subsets of $A_{dom}$ that *satisfy* the constraint. A constraint $A$ is said to be *monotone* if for every $X \subseteq Y \subseteq A_{dom}$, $X \in A_{sat}$ implies that $Y \in A_{sat}$. A constraint $A$ is said to be *convex* if for all $X \subseteq Y \subseteq Z \subseteq A_{dom}$ such that $X, Z \in A_{sat}$, $Y \in A_{sat}$, too.

Constraints are building blocks of rules and programs. A *rule* is an expression

$$A \leftarrow A_1, \ldots, A_k \tag{1}$$

where $A$, $A_1, \ldots, A_k$ are constraints. A *constraint program* (or a *program*) is a collection of rules. A program is *monotone* (*convex*) if every constraint occurring in it is monotone (convex).

Given a rule $r$ of the form (1), the constraint $A$ is the *head* of $r$ and the set $\{A_1, \ldots, A_k\}$ of constraints is the *body* of $r$ (sometimes we view the body of a rule as the *conjunction* of its constraints). We denote the head and the body of $r$ by $hd(r)$ and $bd(r)$, respectively. We define the *headset* of $r$, written $hset(r)$, as the domain of the head of $r$. That is, $hset(r) = hd(r)_{dom}$.

We view subsets of $At$ as interpretations. We say that $M$ *satisfies* a constraint $A$, denoted by $M \models A$, if $M \cap A_{dom} \in A_{sat}$. The satisfiability relation extends in the standard way to conjunctions of constraints, rules and programs.

Let $M \subseteq At$ be an interpretation. A rule is $M$*-applicable* if $M$ satisfies every constraint in $bd(r)$. We denote with $P(M)$ the set of all $M$-applicable rules in $P$. Let $P$ be a program. A model $M$ of $P$ is *supported* if $M \subseteq hset(P(M))$.

Let $P$ be a program and $M$ a set of atoms. A set $M'$ is *non-deterministically one-step provable* from $M$ by means of $P$, if $M' \subseteq hset(P(M))$ and $M' \models hd(r)$ for every rule $r \in P(M)$. The *nondeterministic one-step provability operator* $T_P^{nd}$ for a program $P$ is an operator on $\mathcal{P}(At)$ such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists of all sets that are non-deterministically one-step provable from $M$ by means of $P$.

For an arbitrary atom $a \in At$, the constraint $(\{a\}, \{\{a\}\})$ is called an *elementary constraint*. Since $(\{a\}, \{\{a\}\})$ has the same models as $a$, $(\{a\}, \{\{a\}\})$ is often identified with (and denoted by) $a$. For the same reason, $(\{a\}, \{\emptyset\})$ can be identified with $\mathbf{not}\ a$. Given a normal logic program $P$, by $C(P)$ we denote the program with constraints obtained from $P$ by replacing every positive atom $a$ in $P$ with the constraint $(\{a\}, \{\{a\}\})$, and replacing every literal $\mathbf{not}\ a$ in $P$ with the constraint $(\{a\}, \{\emptyset\})$.

We note that $C(P)$ is a convex program [11]. One can show that supported models of $P$ coincide with supported models of $C(P)$, and answer sets of $P$ coincide with answer sets of $C(P)$ (according to the definition from [11]). In other words, programs with constraints are sufficient to express normal logic programs. Therefore, in this paper we focus on programs with abstract constraints only.

## 4 Computations for Programs with Constraints

Our goal is to extend the concept of a computation to programs with constraints. Once we have such a concept in hand, we will use it to define answer sets for programs with constraints. To this end, we will build on the two characterizations of answer sets for the case of normal logic programs, which we developed in Section 2.

In order to define computations of programs with constraints, we consider the principles identified in Sect. 2. The key step is to generalize the revision principle. For normal programs, it was based on sets of atoms grounded in a set of atoms $X$ (current interpretation) and $P$. We will now extend this concept to programs with constraints.

**Definition 3.** *Let $P$ be a program with constraints and let $X \subseteq At$ be a set of propositional atoms. A set $Y$ is grounded in $X$ and $P$ if for some program $Q \subseteq P(X)$, $Y \in T_Q^{nd}(X)$. We denote by $Concl_P(X)$ the set of all sets $Y$ grounded in $X$ and $P$.*

The intuition is the same as before: a set $Y$ is grounded in $X$ and $P$ if it can be justified by means of some rules in $P$ on the basis of $X$. It follows directly from the definition that if $Q \subseteq P$ then $T_Q^{nd}(X) \subseteq Concl_P(X)$, which generalizes a similar property in the case $P$ is a normal logic program: if $Q \subseteq P$ then $T_Q(X) \in Concl_P(X)$.

With this definition of $Concl_P(X)$, the principle $(\mathbf{R})$ lifts without any changes. The same is true for the principle $(\mathbf{P})$ (which is independent of the choice of the class of programs). The principle $(\mathbf{C})$ is also easy to generalize thanks to its alternative statement in terms of models:

$(\mathbf{C})$ *Convergence*: $X_\infty$ is a supported model of $P$, i.e., $X_\infty \in T_P^{nd}(X_\infty)$.

Finally, the principle $(\mathbf{Pr})$ generalizes, as well. At a step $i$ of a computation that satisfies $(\mathbf{R})$, we select as $X_i$ an element of $Concl_P(X_{i-1})$. By the definition of $Concl_P(X_{i-1})$, there is a program $P_{i-1} \subseteq P(X_{i-1})$ such that $X_i \in T_{P_{i-1}}^{nd}(X_{i-1})$. Each such program can be viewed as a *reason* for $X_i$. We can now state the generalized principle $(\mathbf{Pr})$ as follows:

$(\mathbf{Pr})$ *Persistence of Reasons*: There is a sequence of programs $\langle P_i \rangle_{i=0}^{\infty}$ such that for every $i$, $0 \leq i$, $P_i \subseteq P_{i+1}$, $P_i \subseteq P(X_i)$, $X_{i+1} \in T_{P_i}^{nd}(X_i)$.

Having extended the principles $(\mathbf{R})$, $(\mathbf{P})$, $(\mathbf{C})$ and $(\mathbf{Pr})$ to the class of programs with constraints, we define computations literally extending the earlier definitions.

**Definition 4.** *Let $P$ be a program with abstract constraints. A sequence $\langle X_i \rangle_{i=0}^{\infty}$ is a* computation *for $P$ if $X_0 = \emptyset$ and the sequence satisfies the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$. A computation is* persistent *if it also satisfies the principle $(\mathbf{Pr})$.*

As before, we have the following containment property.

**Proposition 8.** *Let $P$ be a program with constraints. The class of persistent computations is a* proper *subset of the class of computations.*

*Proof (Sketch).* The containment is evident. To show that it is proper, we consider the program $C(P_2)$, where $P_2$ is the normal logic program from Example 2. The sequence $\emptyset, \{a\}, \ldots$ is a computation but not a persistent computation for $C(P_2)$.

It is also the case, that computations for programs with constraints generalize computations for normal logic programs.

**Proposition 9.** *Let $P$ be a normal logic program. The class of computations (respectively, persistent computations) of $P$, according to the definitions in Section 2, coincides with the class of computations (respectively, persistent computations) of the program $C(P)$, according to definitions in Section 3.*

We conclude this section by proposing the first definition of the concept of *answer set* for programs with constraints. To this end, we generalize the characterization of answer sets of normal logic programs in terms of persistent computations discussed in Section 2.

**Definition 5.** *Let $P$ be a program with constraints. A set $X$ is an* answer set *of $P$ if there is a persistent computation for $P$, whose result is $X$.*

Since persistent computations satisfy the principle $(\mathbf{C})$, answer sets of a program $P$ with constraints are supported models of $P$, generalizing a property of normal logic programs. As a matter of fact, the results of arbitrary computations are supported models (because of $(\mathbf{C})$). However, there are programs such that some of their supported models cannot be reached by a computation. For instance, $\{a\}$ is a supported model of the program $C(P)$, where $P = \{a \leftarrow a\}$, but there is no computation for $C(P)$ with the result $\{a\}$.

Proposition 9 implies that our definition of answer sets for programs with constraints generalizes the definition of answer sets for normal logic programs.

**Corollary 1.** *Let $P$ be a normal logic program. A set $X \subseteq At$ is an answer set of $P$ if and only if $X$ is an answer set of $C(P)$.*

It is also the case that this concept of answer sets extends that introduced in [11, 16] for monotone and convex programs.

**Proposition 10.** *Let $P$ be a monotone or convex program. Then, a set of atoms $X \subseteq At$ is an answer set of $P$ according to the definition in [11, 16] if and only if $X$ is the answer set of $P$ according to Definition 5.*

## 5 Computations and Quasi-Satisfiability Relations

The notion of a computation discussed so far makes use of the non-deterministic operator $Concl_P$ to revise the interpretations occurring along a computation. As we mentioned earlier, the use of $Concl_P$ provides a wide range of choices for revising a state of a computation, essentially considering all the subsets of applicable rules.

We will now study computations, as well as related concepts resulting by relaxing some of the postulates for computations, which can be obtained by narrowing down the set of choices given by $Concl_P(X)$ as possible revisions of $X$. In the case of normal logic programs, we accomplished this goal by means of an operator $T_P^M$, based on the satisfiability relation $\models_M$. We will now generalize that idea to the case of programs with constraints.

**Definition 6.** *A sequence* $C = \langle X_i \rangle_{i=0}^{\infty}$ *is a* weak computation *for a program with constraints,* $P$*, if* $X_0 = \emptyset$ *and if* $C$ *satisfies the properties* $(\mathbf{P})$ *and* $(\mathbf{C})$.

Thus, weak computations are sequences that do not rely on a program $P$ when moving from step $i$ to step $i+1$. We will now define a broad class of weak computations that, at least to some degree, restore the role of $P$ as a revision mechanism.

Let $\triangleright$ be a relation between sets of atoms (interpretations) and abstract constraints. We extend the relation $\triangleright$ to the case of conjunctions (sets) of constraints as follows: $X \triangleright A_1, \ldots, A_k$ if $X \triangleright A_i$, for every $i$, $1 \leq i \leq k$. This relation is intended to represent some concept of satisfiability of constraints and their conjunctions. We will call such relations *quasi-satisfiability* relations. They will later allow us to generalize the relation $\models_M$.

For a quasi-satisfiability relation $\triangleright$, we define

$$P^{\triangleright}(X) = \{r \in P \mid X \triangleright bd(r)\}.$$

In other words, $P^{\triangleright}(X)$ is the set of all rules in $P$ that are applicable with respect to $X$ under the relation $\triangleright$. Next, we define $T_P^{nd;\triangleright}(X)$ to consist of all sets $Y \subseteq hset(P^{\triangleright}(X))$ such that $Y \models hd(r)$, for every $r \in P^{\triangleright}(X)$. In other words $T_P^{nd;\triangleright}$ works similarly to $T_P^{nd}$, except that rules in $P^{\triangleright}(X)$ are "fired" rather than those in $P(X)$.

**Definition 7.** *Let* $\triangleright$ *be a quasi-satisfiability relation. A weak computation* $C = \langle X_i \rangle_{i=0}^{\infty}$ *is a* $\triangleright$-weak computation *for* $P$ *if* $X_i \in T_P^{nd;\triangleright}(X_{i-1})$*, for* $i \geq 1$.

Since we do not impose any particular properties on $\triangleright$, it is not guaranteed that $T_P^{nd;\triangleright}(X) \subseteq Concl_P(X)$. Thus, $\triangleright$-weak computations are not guaranteed to be computations.

We say that a quasi-satisfiability relation $\triangleright$ is a *sub-satisfiability* relation if for every $X \subseteq At$ and every abstract constraint $A$, $X \triangleright A$ implies $X \models A$.

We note that relations $\models_M$ considered in Section 2 are sub-satisfiability relations (with respect to the standard satisfiability relation $\models$).

**Proposition 11.** *Let* $P$ *be a program with constraints. If* $\triangleright$ *is a sub-satisfiability relation then for every* $X \subseteq At$*,* $T_P^{nd;\triangleright}(X) \subseteq Concl_P(X)$ *and every* $\triangleright$*-weak computation is a computation.*

From now on, if $\triangleright$ is a sub-satisfiability relation, we will write $\triangleright$*-computation* instead of $\triangleright$-weak computation. We will now define another class of answer sets for programs with constraints.

**Definition 8.** *Let* $P$ *be a program with constraints and* $\triangleright$ *a sub-satisfiability relation. Then,* $M$ *is a* $\triangleright$-answer set *of* $P$ *if* $M$ *is the result of a persistent* $\triangleright$*-computation.*

Since $\triangleright$-computations are computations, we have the following direct consequence of the appropriate definitions.

**Proposition 12.** *Let* $P$ *be a program with constraints and* $\triangleright$ *a sub-satisfiability relation. Then every* $\triangleright$*-answer set for* $P$ *is an answer set for* $P$.

A natural questions arises whether every answer set of a program with constraints is a $\triangleright$-answer set of that program for some sub-satisfiability relation $\triangleright$. Unlike in the case of normal logic programs, it is not the case.

*Example 4.* Let $P$ consist of three rules:

$$(\{a\},\{\{a\}\}). \quad (\{b\},\{\{b\}\}). \quad (\{c\},\{\{c\}\}) \leftarrow (\{a,b,c\},\{\{a\},\{a,c\},\{a,b,c\}\}).$$

We first note that $X_0 = \emptyset$, $X_1 = \{a\}$, $X_2 = \{a,c\}$, $X_i = \{a,b,c\}$, $i \geq 3$, is a persistent computation for $P$. Thus, $\{a,b,c\}$ is an answer set or $P$. However, there is no sub-satisfiability relation $\triangleright$ such that $\{a,b,c\}$ is a $\triangleright$-answer set for $P$. Indeed, for each such relation $\triangleright$ we have $T_P^{nd;\triangleright}(\emptyset) = \{a,b\}$, and it is impossible to derive $c$, as the third rule is not applicable with respect to $\{a,b\}$ (and so, also not a member of $P^{\triangleright}(\{a,b\})$). $\square$

Thus, given a program $P$, the class of $\triangleright$-answer sets is a proper subset of the class of answer sets of $P$.

The class of $\triangleright$-answer sets forms a generalization of answer sets of normal logic programs given by Proposition 5. We will now propose a way to generalize answer sets of normal logic programs to programs with constraints based on Proposition 7. In our considerations we extend the approach proposed and studied in [21]. Our method requires a fixed mapping $f$ that assigns to each weak computation $C$ a quasi-satisfiability relation $\triangleright_C^f$. For some mappings $f$ it yields models that are not "grounded enough" to be called answer sets. For some other mappings $f$, however, it does result in classes of models that arguably generalize answer sets of normal logic programs.

**Definition 9.** *Let $P$ be a program with constraints and $f$ a mapping assigning to each weak computation $C$ a quasi-satisfiability relation $\triangleright_C^f$. A weak computation $C$ is $(P,f)$* self-justified *if $C$ is a $\triangleright_C^f$-weak computation for $P$. A set of atoms $M$ is an* $f$-model *of $P$ if $M$ is the result of a $(P,f)$ self-justified weak computation.*

The definition of an $f$-model is sound. Since weak computations satisfy the property $(\mathbf{C})$, their results are indeed models of $P$, in fact, even supported models.

Several interesting classes of models of programs with constraints can be described in terms of $f$-models by specializing the mapping $f$.

**Supported models.** Let $C$ be a weak computation. We define the relation $\triangleright_C^{supp}$ as follows: given a set of atoms $X$ and a constraint $A$, $X \triangleright_C^{supp} A$ if $X_\infty \models A$. One can show that for every supported model $M$ of $P$, the sequence $C = \langle \emptyset, M, M, \ldots \rangle$ is a weak computation self-justified with respect to $P$ and $\triangleright_C^{supp}$. Thus, every supported model of $P$ is a $supp$-model of $P$. As we observed earlier, all $f$-models are supported models. It follows that supported models of $P$ are precisely $supp$-models of $P$.

***Mr*-models.** Let $C$ be a weak computation. We define the relation $\triangleright_C^{mr}$ as follows: given a set of atoms $X$ and a constraint $A$, $X \triangleright_C^{mr} A$ if there is $Y \subseteq X$ such that $Y \models A$ and $X_\infty \models A$. One can show that $mr$-models of $P$ are precisely the answer sets of $P$ as defined by [15].

The discussion of $supp$-models and $mr$-models was meant to show the flexibility of our approach. It took us away, however, from the main theme of the paper — generalizations of the concept of an answer set. Indeed, neither $\triangleright_C^{supp}$-weak computations nor

$\triangleright_C^{mr}$-weak computations are computations (they do not satisfy the revision principle). Therefore, we do not view their results as "generalized" answer sets but only as some special classes of models.

To specialize the general approach of self-justified weak computations so that it yields extensions of the concept of an answer set, we need to look for mappings $f$ that ensure that self-justified weak computations are computations (satisfy the revision principle) and are persistent.

We already saw that requiring that $\triangleright_C^f$ be a sub-satisfiability relation guarantees that $\triangleright_C^f$-weak computations are indeed computations (referred to, we recall, as $\triangleright_C^f$-computations). We will now seek conditions guaranteeing the persistence of $\triangleright_C^f$-computations.

Under the assumption that $\triangleright$ is a sub-satisfiability relation, $P^{\triangleright}(X) \subseteq P(X)$. This property and the appropriate definitions imply that

$$T_P^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd}(X).$$

Consequently, we can show the following result.

**Proposition 13.** *Let $\triangleright$ be a sub-satisfiability relation and let $C = \langle X_i \rangle_{i=0}^{\infty}$ be a $\triangleright$-computation. If for every constraint $A$ and every $i = 0, 1, \ldots$, $X_i \triangleright A$ implies that $X_{i+1} \triangleright A$, then $C$ is persistent.*

We will now define the mapping $s$, which assigns to every weak computation $C$ a relation $\triangleright_C^s$. Namely, for a set of atoms $X$ and a constraint $A$ we define $X \triangleright_C^s A$ if there is $i$ such that $X = X_i$ and $X_j \models A$, for every $j \geq i$. It is clear that $\triangleright_C^s$ is a sub-satisfiability relation. Thus, $\triangleright_C^c$-weak computations are computations ($\triangleright_C^s$-computations, to be precise). Moreover, it follows from Proposition 13 that $\triangleright_C^s$-computations are persistent.

**Definition 10.** *Let $P$ be a program with constraints. A set of atoms $M$ is a* strong answer set *of $P$ (or, an* s-answer *set for $P$) if it is an s-model for $P$, that is, if $M$ is the result of a $(P, s)$ self-justified computation.*

We will now summarize our discussion so far. Taking characterizations of answer sets of normal logic programs as the starting point, we proposed three notions of answer sets for programs with constraints. For normal logic programs, for programs with monotone constraints, and for programs with convex constraints all three concepts coincide with the standard notion of an answer set. For general programs with constraints, these three concepts are different. The following results summarizes the relationships between the three classes of answer sets we introduced so far.

**Proposition 14.** *Let $P$ be a program with constraints. The class of strong answer sets for $P$ is a proper subset of the class of $\triangleright$-answer-sets for $P$ which, in turn, is a proper subset of the class of answer sets for $P$.*

*Example 5.* Consider the program $P$ (remember that $a$ is shorthand for $(\{a\}, \{\{a\}\})$):

$$\begin{array}{l} a \leftarrow (\{a,b,c,d\}, \mathcal{P}(\{a,b,c,d\})) \\ b \leftarrow (\{a,b\}, \{\{a\},\{a,b\}\}) \\ c \leftarrow (\{a,b,c,d\}, \{\emptyset, \{a\}, \{a,b\}, \{a,b,c,d\}\}) \end{array}$$

where $\mathcal{P}(X)$ is the powerset of $X$. Let us define $X \triangleright A$ if for every $Y$, $X \cap A_{dom} \subseteq Y \subseteq A_{dom}$, $Y \in A_{sat}$. Clearly, $\triangleright$ is a sub-satisfiability relation. Furthermore $\emptyset$, $\{a\}$, $\{a,b\}, \{a,b\} \ldots$ is a $\triangleright$-weak computation, say $C$. On the other hand, we have that $\emptyset \not\triangleright^s_C (\{a,b,c,d\}, \mathcal{P}(\{a,b,c,d\}))$ and $\emptyset \triangleright^s_C (\{a,b,c,d\}, \{\emptyset, \{a\}, \{a,b\}, \{a,b,c,d\}\})$. Thus, any $(P,s)$ self-justified computation will need to have $\{a,c\}$ as its second element and will be unable to reach $\{a,b\}$. This shows that $\{a,b\}$ is not a strong answer set of $P$. We note that it follows from Example 4 that the second inclusion is proper. $\square$

## 6 Yet another class of answer sets

Given a computation $C$, we defined the relation $\triangleright^s_C$ so that it is the weakest sub-satisfiability relation satisfying the assumptions of Proposition 13. However, in general there may be other ways to define a sub-satisfiability relation $\triangleright_C$ with respect to a given computation $C$. One such definition was proposed in [21]. Namely, given a weak computation $C = \langle X_i \rangle_{i=0}^{\infty}$, we define $\triangleright^{spt}_C$ as follows: $X \triangleright^{spt}_C A$ if $X \models A$ and for each set $Y$ such that $X \cap A_{dom} \subseteq Y \subseteq X_\infty \cap A_{dom}$ we have that $Y \models A$ (or equivalently, $Y \in A_{sat}$). It is easy to see that $\triangleright^{spt}_C$ is a sub-satisfiability relation. Thus, it defines computations. Secondly, $\triangleright^{spt}_C$-computations are persistent as the relation $\triangleright^{spt}_C$ satisfies the assumptions of Proposition 13. Thus, the mapping $spt$ gives rise to yet another class of answer sets — $spt$-answer sets. One can show that spt-answer sets capture precisely the semantics for aggregates proposed in [18, 19].

We have the following result relating $spt$-answer sets to other classes of answer sets considered in the previous section.

**Proposition 15.** *Let $P$ be a program with constraints. If a computation $C$ is a $\triangleright^{spt}_C$-computation then it is also a $\triangleright^s_C$-computation.*

*Example 6.* Let us consider the program $P$ consisting of two rules

$$(\{b\}, \{\{b\}\}) \leftarrow (\{a\}, \{\{a\}\}) \qquad (\{a\}, \{\{a\}\}) \leftarrow (\{a,b\}, \{\emptyset, \{a\}, \{a,b\}\})$$

The sequence $\emptyset$, $\{a\}$, $\{a,b\}$ is a $\triangleright^s_C$-computation of $P$. On the other hand, it is not a $\triangleright^{spt}_C$ computation since $\emptyset \not\triangleright^{spt}_C (\{a,b\}, \{\emptyset, \{a\}, \{a,b\}\})$. $\square$

**Corollary 2.** *Let $P$ be a program with constraints. If $M$ is an spt-answer set of $P$ then $M$ is a strong answer set of $P$.*

Next, we note that, similarly to our other classes of answer sets, the semantics defined by $spt$-answer sets also collapses to the standard answer-sets semantics for normal logic programs and to the answer-sets semantics for programs with convex constraints [11].

The approach based on the mapping $spt$ takes a more "skeptical" perspective than the one embodied by the mapping $s$. To ensure persistence, it requires that *all* possible extensions of the state in which a constraint $A$ holds satisfy $A$ (and not only those actually encountered during the computation). In this way, the relations $\triangleright^{spt}_C$ are, in a sense, the strongest relations satisfying the assumptions of Proposition 13. This may be perceived as a problem of this semantics — $\triangleright^{spt}_C$-computations are localized to convex subfragments of constraints forming program rules. In other words, they cannot jump over "missing" satisfiers.

## 7 A Note on the Complexity

In this paper we introduced several classes of answer sets for programs with constraints. We have the following result concerning the computational complexity of the problem concerning the existence of answer sets.

**Proposition 16.** *Assuming an explicit representation of constraints, given a program with constraints, it is NP-complete to decide whether the program has an answer set (respectively, $\triangleright$-answer set, strong answer set, $spt$-answer set).*

## 8 Discussion and Conclusions

We grounded our study in four basic principles: revision, persistence of beliefs, persistence of reasons, and convergence. We showed that there are several ways in which the principle of revision can be realized. In a least restrictive approach, we allow any element of $Concl_P(X)$ to be a valid revision of $X$. This choice defines the class of persistent computations and we take the results of such computations as the definition of the first notion of an answer set. In the case of normal logic programs and programs with convex constraints, computations capture precisely the concept of an answer sets as defined for these classes of programs earlier in [8, 11].

More restrictive approaches to the revision principle narrow down choices offered by $Concl_P$ to those offered by $T_P^{nd;\triangleright}$, where $\triangleright$ is a sub-satisfiability relation. The results of persistent $\triangleright$-computations form another class of answer sets, $\triangleright$-answer sets, which forms a proper subclass of the previous one. However, in the case of normal logic programs and programs with convex constraints both notions of the answer set coincide.

The final two approaches result from the two specializations of a general schema to define $f$-models of a program with constraints. The schema is designed to generalize the guess-and-check approach for normal logic programs. It relies on a mapping that assigns to weak computations quasi-satisfiability relations. We demonstrated two mappings, $s$ and $spt$, for which the resulting weak computations are in fact persistent computations and so, their results can be used as answer sets. The mapping $s$ seems to be more appropriate as it is less restrictive. The mapping $spt$, on the other extreme of the spectrum, seems to be too restrictive. As we noted earlier programs that intuitively should have answer sets do not have $spt$-answer sets.

This work draws attention to the concept of computation, and shows that, for programs with arbitrary constraints, there are many classes of computations of interest. In general, they give rise to different classes of answer set, by formalizing in different ways the negation-as-failure implicitly present in (non-monotone) constraints. Three classes of computations seem especially well suited as the basis for the generalization. Specifically, $s$-answer sets, $\triangleright$-answer sets and answer sets are viable candidates for the answer set semantics of programs with constraints, as they are grounded in some basic and intuitive principles imposed on computations and on schemata to define computations generalizing those used earlier in the context of normal logic programs. The class of $spt$-answer sets may be too restrictive. The issue whether any of the three classes identified here has any significant advantage over the other two requires further studies.

We note that some of our methods go beyond generalizations of just answer sets. if we weaken requirements on computations and consider the class of weak computations, the general schema of defining $f$-models of programs yields characterizations of supported models and mr-answer sets [15] and so also deserves further attention.

## References


1. M. Balduccini, M. Gelfond, and M. Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 2006.
2. T. Dell'Armi et al. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *IJCAI*, pages 847–852, 2003.
3. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. *ICLP*, Springer Verlag, pages 212–226, 2001.
4. I. Elkabani, E. Pontelli, and T.C. Son. Smodels with CLP and its applications. In *Int. Conference on Logic Programming*, Springer Verlag, pages 73–89, 2004.
5. E. Erdem, V. Lifschitz, and D. Ringe. Temporal phylogenetic networks and logic programming. *TPLP*, 6(5):539–558, 2006.
6. W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *JELIA*, pages 200–212, 2004.
7. P. Ferraris. Answer sets for propositional theories. In *Logic Programming and Nonmonotonic Reasoning*, Springer Verlag, pages 119–131, 2005.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Joint Int. Conf. and Symp. on Logic Programming*, pages 1070–1080, MIT Press, 1988.
9. M. Gelfond. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond*, pages 413–451. Springer Verlag, 2002.
10. K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4,5):519–550, 2003.
11. L. Liu and M. Truszczyński. Properties of programs with monotone and convex constraints. *National Conference on Artificial Intelligence*, pages 701–706. AAAI/MIT Press, 2005.
12. V.W. Marek, A. Nerode and J. Remmel. Logic Programs, Well-orderings, and Forward Chaining. *Annals of Pure and Applied Logic* 96:231-276, 1999.
13. V.W. Marek and M. Truszczyński. *Nonmonotonic Logic; Context-Dependent Reasoning.* Springer Verlag, 1993.
14. V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, pp. 375–398, Springer, 1999.
15. V.W. Marek and J.B. Remmel. Set constraints in logic programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 167–179, Springer Verlag, 2004.
16. V.W. Marek and M. Truszczyński. Logic programs with abstract constraint atoms. In *National Conference on Artificial Intelligence (AAAI)* AAAI Press / The MIT Press, 2004.
17. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
18. N. Pelov. *Semantic of Logic Programs with Aggregates*. PhD thesis, K.U. Leuven, 2004.
19. T.C. Son and E. Pontelli. A Constructive Semantic Characterization of Aggregates in Answer Set Programming. *Theory and Practice of Logic Programming*, 2007.
20. T.C. Son, E. Pontelli, and I. Elkabani. An Unfolding-Based Semantics for Logic Programming with Aggregates. *Computing Research Repository*, 2006. cs.SE/0605038.
21. T.C. Son, E. Pontelli, and P.H. Tu. Answer Sets for Logic Programs with Arbitrary Abstract Constraint Atoms. *Journal of Artificial Intelligence Research*, 2007. Accepted.
22. M.H. van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4), 733–742, 1976.