From 3-valued Semantics to Supported Model Computation for Logic Programs in Vector Spaces*

Taisuke Sato[1][a], Chiaki Sakama[2][b] and Katsumi Inoue[3][c]
[1]National Institute of Advanced Industrial Science and Technology, Japan
[2]Wakayama University, Japan
[3]National Institute of Informatics, Japan

Keywords: Dualized Logic Program, Supported Model, 3-valued, Vector Space.

Abstract: We propose a linear algebraic approach to computing 2-valued and 3-valued completion semantics of finite propositional normal logic programs in vector spaces. We first consider 3-valued completion semantics and construct the least 3-valued model of comp(DB), i.e. the iff (if-and-only-if) completion of a propositional normal logic program DB in Kleene's 3-valued logic which has three truth values $\{\mathbf{t}(\text{true}), \mathbf{f}(\text{false}), \perp(\text{undefined})\}$. The construction is carried out in a vector space by matrix operations applied to the matricized version of a dualized logic program DB^d of DB. DB^d is a definite clause program compiled from DB and used to compute the success set P_∞ as true atoms and finite failure set N_∞ as false atoms that constitute the least 3-valued model $I_\infty = (P_\infty, N_\infty)$ of comp(DB). We then construct a supported model of DB, i.e. a 2-valued model of comp(DB) by carefully assigning $\mathbf{t}$ or $\mathbf{f}$ to undefined atoms not in P_∞ or N_∞ so that the resulting model is 2-valued and supported. The assigning process is guided by an atom dependency relation on undefined atoms. We implemented our proposal by matrix operations and conducted an experiment with random normal logic programs which demonstrated the effectiveness of our linear algebraic approach to computing completion semantics.

1 INTRODUCTION

Performing logical inference in vector spaces has been studied as an attractive alternative to the traditional symbolic inference which opens a way to take advantage of flexible matrix operations in vector spaces and associated parallelism supported by recent computer technologies (Sato, 2017; Sakama et al., 2017; Sato et al., 2018). For example it was demonstrated Datalog programs can be computed orders of magnitude faster in vector spaces than by symbolic computation when relations are not too sparse (Sato, 2017). Also it is possible to invent new relations for more than 10^4 entities from real knowledge graphs by reformulating abduction in vector spaces (Sato et al., 2018). In this paper, along the same line, we develop a linear algebraic approach to computing completion semantics of logic programs in vector spaces.

[a] https://orcid.org/0000-0001-9062-0729
[b] https://orcid.org/0000-0002-9966-3722
[c] https://orcid.org/0000-0002-2717-9122

*This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

We first consider 3-valued completion semantics and construct the least 3-valued model of comp(DB) where comp(DB) is the iff (if-and-only-if) completion of a finite propositional normal logic program DB as the 3-valued denotation of DB, in Kleene's 3-valued logic which is based on three truth values $\{\mathbf{t}(\text{true}), \mathbf{f}(\text{false}), \perp(\text{undefined})\}$ (Fitting, 1985; Kunen, 1987). We prove that the model construction can be deterministically carried out in a vector space by matrix operations applied to the matricized version of a dualized logic program DB^d of DB. DB^d is a finite definite clause program compiled from DB and used to compute DB's success set P_∞ as true atoms, and DB's finite failure set N_∞ as false atoms. They constitute the least 3-valued model $I_\infty = (P_\infty, N_\infty)$ of comp(DB). We then construct a 2-valued model of comp(DB), i.e. supported model (Apt et al., 1988; Marek and V.S.Subrahmanian, 1992) of DB by appropriately assigning $\mathbf{t}$ or $\mathbf{f}$ to undefined atoms not in P_∞ or N_∞ so that the resulting model is 2-valued and supported. Every supported model of DB is obtained this way. The assignment process is guided by an atom dependency relation on undefined atoms and processes them from bottom-layers. Since every true (resp.

Sato, T., Sakama, C. and Inoue, K.
From 3-valued Semantics to Supported Model Computation for Logic Programs in Vector Spaces.
DOI: 10.5220/0009093407580765
In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020) - Volume 2*, pages 758-765
ISBN: 978-989-758-395-7; ISSN: 2184-433X

false) atom in I_∞ remains true (resp. false) in any supported model of DB, we may say that I_∞ represents the "deterministic part" of supported models (or stable models) of DB. To our knowledge, our approach is the first that computes 3-valued completion semantics in vector spaces and also the first that computes the deterministic part of supported models separately.

2 PRELIMINARIES

In this paper, programs mean finite propositional normal logic programs DB made up of a set of atoms $\mathcal{A}$ unless otherwise stated. Write DB = $\{a_i \leftarrow B_i \mid 1 \leq i \leq N, a_i \in \mathcal{A}\}$ where B_i is a conjunction of literals whose atoms are in $\mathcal{A}$[1]. Let $a \leftarrow B_1, \ldots, a \leftarrow B_k$ be clauses about a in DB. Introduce new constant atoms `true` and `false` respectively denoting **t** and **f**. We define the *iff (if-and-only-if) completion* about a as follows.

If $k = 0$, i.e. there is no clause about a, put $\mathrm{iff}(a) = a \Leftrightarrow \texttt{false}$. If $k = 1$ and B_1 is an empty conjunction, i.e. a is a unit clause, put $\mathrm{iff}(a) = a \Leftrightarrow \texttt{true}$. Otherwise put $\mathrm{iff}(a) = a \Leftrightarrow B_1 \vee \cdots \vee B_k$. The *completion of* DB, denoted by comp(DB), is defined as comp(DB) = $\{\mathrm{iff}(a) \mid a \in \mathcal{A}\}$.

We consider 3-valued completion semantics of logic programs following Fitting and Kunen(Fitting, 1985; Kunen, 1987). Their semantics is based on Kleene's 3-valued logic that has three truth values { **t**(true), **f**(false), $\perp$(undefined) }. In Kleene's 3-valued logic, when neither A nor B is $\perp$, conjunction $A \wedge B$, disjunction $A \vee B$ and negation $\neg A$ are evaluated as usual. However, if one of them is $\perp$, we obey the following rules; $A \wedge B = \perp$ iff one conjunct is $\perp$ and the other is not false, $A \vee B = \perp$ iff one disjunct is $\perp$ and the other is not true, and $\neg A = \perp$ iff $A = \perp$. We treat implication $A \leftarrow B$ as $\neg A \vee B$. Accordingly $\perp \leftarrow \perp = \perp$.

A 3-valued interpretation I in $\mathcal{A}$ is a pair (P,N) such that $P \cup N \subseteq \mathcal{A}$ and $P \cap N = \emptyset$ where P stands for true atoms and N for false atoms. We next define $[\![F]\!]_I \in \{\mathbf{t}, \mathbf{f}, \perp\}$, i.e. the truth value of a Boolean formula F evaluated by $I = (P,N)$. We first define $[\![\texttt{true}]\!]_I = \mathbf{t}$ and $[\![\texttt{false}]\!]_I = \mathbf{f}$, then define $[\![F]\!]_I$ for $F = a \in \mathcal{A}$ by $[\![a]\!]_I = \mathbf{t}$, $\mathbf{f}$ and $\perp$ respectively if $a \in P$, $a \in N$ and otherwise and extend the definition to compound Boolean formulas according to Kleene's 3-valued logic.

Since we are interested in comp(DB), in particular in iff(a) = $a \Leftrightarrow B_1 \vee \cdots \vee B_k$ and the equality of truth values of a and $B_1 \vee \cdots \vee B_k$, we deviate here from Kleene's 3-valued logic and interpret $\Leftrightarrow$ in a 2-valued way. We stipulate that $[\![A \Leftrightarrow B]\!]_I = \mathbf{t}$ if $[\![A]\!]_I = [\![B]\!]_I$. Otherwise $[\![A \Leftrightarrow B]\!]_I = \mathbf{f}$. So $[\![\perp \Leftrightarrow \perp]\!]_I = \mathbf{t}$ and $[\![\perp \Leftrightarrow \mathbf{t}]\!]_I = \mathbf{f}$. If $[\![\mathrm{iff}(a)]\!]_I = \mathbf{t}$ holds for every $a \in \mathcal{A}$, we call I a 3-valued model of comp(DB) or equivalently a 3-valued *completion model* of DB[2].

It is known that Fitting's 3-valued semantics of normal logic programs is highly undecidable whereas Kunen's 3-valued one is recursively enumerable (Fitting, 1985; Kunen, 1987), but they coincide in the case of finite propositional normal logic programs, which we describe next. Here we inductively define a series of 3-valued interpretations $I_0 = (P_0, N_0), I_1 = (P_1, N_1), \ldots$ (Sato, 1990).

(base case) $P_0 = N_0 = \emptyset$
(inductive step)
Put $I_{n-1} = (P_{n-1}, N_{n-1})$
$P_n = \{a \in \mathcal{A} \mid \text{there is } a \leftarrow B \text{ in DB s.t. } [\![B]\!]_{I_{n-1}} = \mathbf{t}\ \}$
$N_n = \{a \in \mathcal{A} \mid \text{for all } a \leftarrow B \text{ in DB, } [\![B]\!]_{I_{n-1}} = \mathbf{f}\}$

Figure 1: Defining 3-valued interpretations $\{(P_n, N_n)\}$.

It is straightforward to verify $P_n \cap N_n = \emptyset$ (n=0,1...), and hence $I_n = (P_n, N_n)$ is a 3-valued interpretation. Let (A_1, A_2) and (B_1, B_2) be pairs of sets. We introduce a partial ordering "$\sqsubseteq$" on pairs of sets by $(A_1, A_2) \sqsubseteq (B_1, B_2)$ iff $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$. Now we see $I_0 \sqsubseteq I_1 \sqsubseteq \cdots$ (Sato, 1990). We introduce P_∞ by $P_\infty = \bigcup P_n$ and N_∞ by $N_\infty = \bigcup N_n$ respectively and put $I_\infty = (P_\infty, N_\infty)$.

We list some propositions (see (Sato, 1990) for proofs).

Proposition 1. *I_∞ is a 3-valued model of comp(DB).*

Proposition 2. *I_∞ is the least 3-valued model of comp(DB) in the sense of $\sqsubseteq$-ordering.*

It is apparent that when DB is a definite clause program, P_∞ in $I_\infty = (P_\infty, N_\infty)$ gives the success set of DB, i.e. the least 2-valued model of DB (whereas N_∞ gives the finite failure set of DB). We consider the least 3-valued model I_∞ as the denotation of DB in our 3-valued semantics.

A 2-valued completion model of DB, i.e. a 2-valued interpretation satisfying comp(DB), is called a *supported model* of DB (Apt et al., 1988; Marek and V.S.Subrahmanian, 1992). DB may have no supported model (think of $a \leftarrow \neg a$) but if DB has a supported model represented by a set P_s ($\subseteq \mathcal{A}$), we see $I_\infty \sqsubseteq I_s = (P_s, \mathcal{A} \setminus P_s)$ thanks to Proposition 2 because every 2-valued completion model is also a 3-valued completion model. In other words, every sup-

[1] Throughout this paper, we assume that if a program has a unit clause $a \leftarrow$, there is no other clause about a in the program.

[2] Likewise, a 2-valued model of comp(DB) is called a 2-valued completion model of DB.

ported model is obtained by appropriately assigning **t** or **f** to the undefined atoms in $U_\infty = \mathcal{A} \setminus (P_\infty \cup N_\infty)$. Supported models are a super class of stable models and when DB is *tight* (no looping caller-callee chain through positive goals), DB's supported models and DB's stable models coincide (Erdem and Lifschitz, 2003).

$$\mathrm{DB}_0 \;=\; \left\{ \begin{array}{l} \mathtt{a} \leftarrow \neg\mathtt{b} \wedge \mathtt{c} \\ \mathtt{b} \leftarrow \neg\mathtt{a} \wedge \mathtt{c} \\ \mathtt{c} \leftarrow \neg\mathtt{d} \end{array} \right.$$

$$\mathrm{comp}(\mathrm{DB}_0) \;=\; \left\{ \begin{array}{l} \mathtt{a} \Leftrightarrow \neg\mathtt{b} \wedge \mathtt{c} \\ \mathtt{b} \Leftrightarrow \neg\mathtt{a} \wedge \mathtt{c} \\ \mathtt{c} \Leftrightarrow \neg\mathtt{d} \\ \mathtt{d} \Leftrightarrow \mathtt{false} \end{array} \right.$$

Figure 2: A program DB_0 and its completion $\mathrm{comp}(\mathrm{DB}_0)$.

Look at DB_0 in Figure 2. We see $(P_0,N_0) = (\emptyset,\emptyset)$, $(P_1,N_1) = (\emptyset,\{\mathtt{d}\})$, $(P_2,N_2) = (\{\mathtt{c}\},\{\mathtt{d}\})$ and $(P_3,N_3) = (P_2,N_2) = I_\infty$. Then $\{\mathtt{a},\mathtt{b}\}$ are undefined atoms U_∞ in I_∞. So any supported model of DB_0 is obtained by assigning **t** or **f** to each of a and b appropriately. Actually $(\{\mathtt{a},\mathtt{c}\},\{\mathtt{b},\mathtt{d}\})$ and $(\{\mathtt{b},\mathtt{c}\},\{\mathtt{a},\mathtt{d}\})$ exhaust all supported models of DB_0.

3 MATRICIZING 3-VALUED SEMANTICS

We here compute $I_\infty = (P_\infty,N_\infty)$ in a vector space by matrix. As a preprocessing step, we standardize a program first and then represent the standardized program by a 0-1 matrix.

3.1 Standardization

Let DB be a program, $\mathcal{A}$ the set of all atoms appearing in DB. For every atom $a \in \mathcal{A}$, do the following.

- If a has no clause about it, add $a \leftarrow \mathtt{false}$ to DB.
- If a has a unit clause $a \leftarrow$, replace it with $a \leftarrow \mathtt{true}$.
- If there is more than one clause $\{a \leftarrow B_1,\ldots,a \leftarrow B_k\}$ $(k > 1)$ in DB, replace them with a set of new clauses $\{a \leftarrow b_1 \vee \cdots \vee b_k, b_1 \leftarrow B_1,\ldots,b_k \leftarrow B_k\}$ containing new atoms $\{b_1,\ldots,b_k\}$.

Call the resulting program a *standardized program* of DB. In general, if a program is standardized, every atom a in $\mathcal{A}$ has exactly one clause $a \leftarrow B$ about it and B is true, false, a conjunction or disjunction of literals in $\mathcal{A}$.

Proposition 3. *Let DB′ be a standardized program of DB. Then comp(DB′) and comp(DB) have the same 3-valued completion models as far as atoms in $\mathcal{A}$ are concerned (proof omitted).*

Proposition 3 tells us that to compute a 3-valued model of comp(DB), we may assume DB is standardized. So, hereafter, we only deal with standardized programs.

3.2 Dualized Logic Programs

Here we introduce dualized programs for later use. Let $\mathrm{DB} = \{a_i \leftarrow B_i \mid 1 \leq i \leq N\}$ be a standardized program in a set of atoms $\mathcal{A} = \{a_1,\ldots,a_N\}$. We define a definite clause program DB^d called a *dualized program* of DB that can compute DB's least 3-valued completion model $I_\infty = (P_\infty,N_\infty)$ as its least 2-valued model.

First introduce a set of new atoms $\bar{\mathcal{A}} = \{na_1,\ldots,na_N\}$. The idea is that na_i represents $\neg a_i$ by negation-as-failure (NAF), i.e. an SLD derivation with NAF for a_i finitely fails. We now define a syntactic function $n(\cdot)$ as follows. First put $n(\mathtt{true}) = \mathtt{false}$ and $n(\mathtt{false}) = \mathtt{true}$. Next consider literals. Let l be a literal whose atom is in $\mathcal{A}$. If l is an atom a, define $n(l) = na$. Otherwise l is a negated atom $\neg a$ and put $n(l) = a$. We extend $n(\cdot)$ to conjunction and disjunction by defining $n(l_1 \wedge \cdots \wedge l_m) = n(l_1) \vee \cdots \vee n(l_m)$ and $n(l_1 \vee \cdots \vee l_m) = n(l_1) \wedge \cdots \wedge n(l_m)$. Also we apply $n(\cdot)$ to a set S by $n(S) = \{n(a) \mid a \in S\}$.

Dually we introduce a function $p(\cdot)$ s.t. $p(\mathtt{true}) = \mathtt{true}$ and $p(\mathtt{false}) = \mathtt{false}$. For atoms in $\mathcal{A}$, if l is an atom a, put $p(l) = a$. Otherwise $l = \neg a$ and put $p(l) = na$. $p(\cdot)$ is extended to conjunction and disjunction by $p(l_1 \wedge \cdots \wedge l_m) = p(l_1) \wedge \cdots \wedge p(l_m)$ and $p(l_1 \vee \cdots \vee l_m) = p(l_1) \vee \cdots \vee p(l_m)$.

Finally define a *dualized program* DB^d of DB. It is a definite clause program and defined by $\mathrm{DB}^d = \{a \leftarrow p(B), na \leftarrow n(B) \mid a \leftarrow B \in \mathrm{DB}\}$. Note that if DB is standardized, so is DB^d.

Look at DB_1 in Figure 3. It is a standardization of DB_0 in Figure 2. Notice that the least 2-valued model of DB_1^d is $\{\mathtt{c},\mathtt{nd}\}$ and it exactly corresponds to the least 3-valued model $(\{\mathtt{c}\},\{\mathtt{d}\})$ of $\mathrm{comp}(\mathrm{DB}_1)$. This is not a coincidence. We next prove that DB^d computes the least 3-valued model I_∞ of comp(DB) as its least 2-valued model.

Put $\bar{\mathcal{A}} = \{na \mid a \in \mathcal{A}\}$ and $\mathcal{A}^d = \mathcal{A} \cup \bar{\mathcal{A}}$. Equate $J \subseteq \mathcal{A}^d$ with a 2-valued interpretation s.t. $[\![c]\!]_J = \mathbf{t}$ iff $c \in J$ for an atom $c \in \mathcal{A}^d$. Now define a series of 2-valued interpretations $\{J_n (\subseteq \mathcal{A}^d)\}$ for DB^d by $J_0 = \emptyset$ and $J_n = \{c \mid c \leftarrow D \in \mathrm{DB}^d, [\![D]\!]_{J_{n-1}} = \mathbf{t}\}$. We have $\emptyset = J_0 \subseteq J_1 \subseteq \ldots$ and $J_\infty = \bigcup_n J_n$ gives the least 2-valued model of DB^d as usual. We call $\{J_n\}$ the

$$\mathrm{DB}_1 = \begin{cases} \mathtt{a} \leftarrow \neg\mathtt{b} \wedge \mathtt{c} \\ \mathtt{b} \leftarrow \neg\mathtt{a} \wedge \mathtt{c} \\ \mathtt{c} \leftarrow \neg\mathtt{d} \\ \mathtt{d} \leftarrow \mathtt{false} \end{cases}$$

$$\mathrm{DB}_1{}^d = \begin{cases} \mathtt{a} \leftarrow \mathtt{nb} \wedge \mathtt{c} \\ \mathtt{b} \leftarrow \mathtt{na} \wedge \mathtt{c} \\ \mathtt{c} \leftarrow \mathtt{nd} \\ \mathtt{d} \leftarrow \mathtt{false} \\ \mathtt{na} \leftarrow \mathtt{b} \vee \mathtt{nc} \\ \mathtt{nb} \leftarrow \mathtt{a} \vee \mathtt{nc} \\ \mathtt{nc} \leftarrow \mathtt{d} \\ \mathtt{nd} \leftarrow \mathtt{true} \end{cases}$$

Figure 3: Standardized program DB_1 and its dual program DB_1^d.

defining interpretations associated with J_∞. The following theorem is proved.

Theorem 1. *Suppose DB is a standardized normal logic program and DB^d is the dualized program of DB. Let $I_\infty = (P_\infty, N_\infty)$ be the least 3-valued model of comp(DB) and J_∞ the least 2-valued model of DB^d respectively. Then we have $(P_\infty, n(N_\infty)) = (\mathcal{A} \cap J_\infty, \bar{\mathcal{A}} \cap J_\infty)$ (proof omitted).*

3.3 Matricized Logic Programs

Theorem 1 tells us that the least 3-valued model $I_\infty = (P_\infty, N_\infty)$ of comp(DB) can be computed as the least 2-valued model J_∞ of DB^d. Here we point out that J_∞ is conveniently computable in a vector space using matrix operations. Our linear algebraic approach to completion semantics is motivated to exploit rapidly growing computer power enabled by modern parallel computation technologies such as multicores and GPUs.

Let $\mathrm{DB} = \{a_1 \leftarrow B_1, \ldots, a_N \leftarrow B_N\}$ be a standardized normal logic program where the bodies B_i's are either `true`, `false`, a conjunction or disjunction of atoms in $\mathcal{A} = \{a_1, \ldots, a_N\}$. Write the dualized program as $\mathrm{DB}^d = \{a_1 \leftarrow B_1, \ldots, a_N \leftarrow B_N, na_1 \leftarrow nB_1, \ldots, na_N \leftarrow nB_N\}$. We represent DB^d in terms of a $(2N \times 2N)$ 0-1 matrix Q and a $(2N \times 1)$ *threshold vector* θ[3]. Q is said to be a *matricized* DB^d or a *program matrix* for DB^d. We construct Q and θ by a procedure in Figure 4[4].

[3] Q only stores information about occurrences of atoms in the clause body and does not make a distinction between conjunction and disjunction. So we need to record supplementary information θ to recover the original DB^d.

[4] For a conjunction or disjunction F, we use $|F|$ to denote the number of literals occurring in F.

- Initialize Q to a $(2N \times 2N)$ zero matrix.
- For $i (1 \le i \le N)$, do the following;
  If B_i = `true`, set $Q(i,i) = \theta(i) = 1$.
  If B_i = `false`, set $Q(i+N, i+N) = \theta(i) = 1$.
  Otherwise, let $a_i \leftarrow B_i$ and $na_i \leftarrow nB_i$ be clauses about a_i and na_i in DB^d respectively.
  – If B_i is a conjunction with $|B_i| > 1$, set $\theta(i) = |B_i|$ and $\theta(i+N) = 1$.
  – Otherwise B_i is a disjunction. Set $\theta(i) = 1$ and $\theta(i+N) = |nB_i|$.
  – Write B_i as $l_1 \diamond \cdots \diamond l_m$ $(m > 0)$ where $\diamond$ denotes either $\wedge$ or $\vee$.
  For $p (1 \le p \le m)$,
  If l_p is an atom a_j,
  put $Q(i, j) = Q(i+N, j+N) = 1$.
  Else l_p is a negated atom $\neg a_j$ and
  set $Q(i, j+N) = Q(i+N, j) = 1$.

Figure 4: Constructing a program matrix Q and a threshold vector θ for DB^d.

For i $(1 \le i \le N)$, the body B_i of $a_i \leftarrow B_i$ in DB^d is encoded as the i-th row $Q(i,:)$ of Q whereas the body nB_i of $na_i \leftarrow nB_i$ is encoded as the $i+N$-th row $Q(i+N,:)$ of Q. More specifically, for $k (1 \le k \le 2N)$, when $\theta(k) > 1$, $Q(k,:)$ represents a conjunction of atoms in $\mathcal{A}^d$. However, if $\theta(k) = 1$, $Q(k,:)$ may represent `true`, `false` or a disjunction. We disambiguate them as follows. If $Q(k,:)$ is a zero vector, $Q(k,:)$ represents $c_k \leftarrow \mathtt{false} \in \mathrm{DB}^d$. If $Q(k,k) = 1$ and ($Q(k+N,:)$ or $Q(k-N,:)$ is a zero vector), $Q(k,:)$ represents $c_k \leftarrow \mathtt{true} \in \mathrm{DB}^d$. Otherwise $Q(k,:)$ is a disjunction of atoms in $\mathcal{A}^d$. Thus DB^d is recoverable from Q and θ.

An (8×8) matrix Q_1 below is a program matrix for DB_1 in Figure 3. As we can see, $\mathtt{a} \leftarrow \mathtt{nb} \wedge \mathtt{c}$ is encoded by $Q_1(1,:) = (0\,0\,1\,0\,0\,1\,0\,0)$, $\mathtt{d} \leftarrow \mathtt{false}$ by $Q_1(4,:) = (0\,0\,0\,0\,0\,0\,0\,0)$, and $\mathtt{nd} \leftarrow \mathtt{true}$ by $Q_1(8,:) = (0\,0\,0\,0\,0\,0\,0\,1)$. Also we set $\theta(1) = |\mathtt{nb} \wedge \mathtt{c}| = 2$ as the threshold associated with a conjunctive clause $\mathtt{a} \leftarrow \mathtt{nb} \wedge \mathtt{c}$ while we set $\theta(5) = 1$ for $\theta(5)$ associated with a disjunctive clause $\mathtt{na} \leftarrow \mathtt{b} \vee \mathtt{nc}$[5].

We compute the least 2-valued model J_∞ of DB^d numerically using Q in a vector space. Let $J (\subseteq \mathcal{A}^d)$ be an interpretation for DB^d. J is represented by a $2N$ dimensional 0-1 vector $\mathbf{u}^{(J)}$ which is constructed as follows. For i $(1 \le i \le N)$, put $\mathbf{u}^{(J)}(i) = 1$ if $a_i \in J$. Otherwise $\mathbf{u}^{(J)}(i) = 0$. Similarly put $\mathbf{u}^{(J)}(i+N) = 1$ if $na_i \in J$ and otherwise $\mathbf{u}^{(J)}(i+N) = 0$.

[5] In this paper, a definite clause whose body is a conjunction (resp. disjunction) is called a conjunctive clause (resp. disjunctive clause).

$$Q_1 = \begin{array}{c} \\ \texttt{a} \\ \texttt{b} \\ \texttt{c} \\ \texttt{d} \\ \texttt{na} \\ \texttt{nb} \\ \texttt{nc} \\ \texttt{nd} \end{array} \begin{array}{c} \begin{array}{cccccccc} \texttt{a} & \texttt{b} & \texttt{c} & \texttt{d} & \texttt{na} & \texttt{nb} & \texttt{nc} & \texttt{nd} \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$\theta_1 = \begin{pmatrix} 2 \\ 2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Figure 5: A program matrix Q_1 and a threshold vector θ_1 for DB_1^d.

We here introduce a thresholding notation $(x)_{\geq\theta}$ parameterized with a real number θ by $(x)_{\geq\theta} = 1$ if $x \geq \theta$. Otherwise $(x)_{\geq\theta} = 0$. We further extend $(x)_{\geq\theta}$ to a vector notation $(\mathbf{u})_{\geq\theta}$ with a threshold vector θ in such a way that $(\mathbf{u})_{\geq\theta}(i) = (\mathbf{u}(i))_{\geq\theta(i)}$ $(1 \leq i \leq n)$ when $\mathbf{u}$ and θ are n dimensional vectors.

Now consider the k-th clause $c_k \leftarrow D_k \in \mathrm{DB}^d$ $(1 \leq k \leq 2N)$ and let $Q(k,:)$ be the k-th row encoding D_k. By construction, $Q(k,:)\mathbf{u}^{(J)}$ gives the number literals in D_k which are true in J. Hence when D_k is a conjunction, D_k is true in J iff $Q(k,:)\mathbf{u}^{(J)} = |D_k| (= \theta(k))$, or equivalently $[\![D_k]\!]_J = (Q(k,:)\mathbf{u}^{(J)})_{\geq\theta(k)}$[6]. Similarly, if D_k is a disjunction, we set $\theta(k) = 1$ and have $[\![D_k]\!]_J = (Q(k,:)\mathbf{u}^{(J)})_{\geq\theta(k)}$. Thus in either case, the clause body D_k of $c_k \leftarrow D_k$ is evaluated by J as $[\![D_k]\!]_J = (Q(k,:)\mathbf{u}^{(J)})_{\geq\theta(k)}$, purely in a vector space in terms of matrix multiplication and thresholding. We summarize the argument so far as

Proposition 4. *Let $DB = \{a_1 \leftarrow B_1, \ldots, a_N \leftarrow B_N\}$ be a standardized normal logic program in atoms $\mathcal{A} = \{a_1, \ldots, a_N\}$, DB^d the dualized logic program of DB, J_∞ the least 2-valued model of DB^d, Q a program matrix and θ a threshold vector for DB^d. Compute a $2N$ dimensional vector $\mathbf{u}_\infty = \mathbf{u}^{(J_\infty)}$. Then $\mathbf{u}_\infty$ satisfies $\mathbf{u}_\infty = (Q\mathbf{u}_\infty)_{\geq\theta}$ (proof omitted).*

Proposition 4 says that $\mathbf{u}_\infty$ is a fixedpoint vector of $f(\mathbf{u}) = (Q\mathbf{u})_{\geq\theta}$ but does not tell us how to compute it, in particular, in a solely linear algebraic way.

Proposition 5. *Let $\{J_n\}$ be the defining interpretations associated with the least model J_∞ of DB^d. Define a series of 0-1 vectors $\{\mathbf{u}_n\}$ by $\mathbf{u}_1 = \mathbf{u}^{(J_1)}$[7] and $\mathbf{u}_{n+1} = (Q\mathbf{u}_n)_{\geq\theta}$. Then $\mathbf{u}_n = \mathbf{u}^{(J_n)}$ for all $n \geq 1$ and $\lim_n \mathbf{u}_n = \mathbf{u}_\infty$ (proof omitted).*

We restate Proposition 5 as the least 3-valued model computation procedure in Figure 6. Given a dualized program DB^d of a standardized normal logic program $\mathrm{DB} = \{a_1 \leftarrow B_1, \ldots, a_N \leftarrow B_N\}$, it computes the least 2-valued model J_∞ of DB^d as a $2N$ dimensional 0-1 vector $\mathbf{u}_\infty$ s.t. $\mathbf{u}_\infty = \mathbf{u}^{(J_\infty)}$, using a program matrix Q and a threshold vector θ for DB^d.

Step 1: Compute $\mathbf{u}_1 = \mathbf{u}^{(J_1)}$
Step 2: Iterate for $n = 1, 2 \ldots$
$\mathbf{u}_{n+1} = (Q\mathbf{u}_n)_{\geq\theta}$ until $\mathbf{u}_{n+1} = \mathbf{u}_n$
Step 3: Return $\mathbf{u}_\infty = \mathbf{u}_n$

Figure 6: Computing $\mathbf{u}_\infty$ by a program matrix Q and a threshold vector θ.

Applying the procedure in Figure 6 to DB_1^d, we see $\mathbf{u}_1 = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 1)^T$, $\mathbf{u}_2 = (Q_1\mathbf{u}_1)_{\geq\theta_1} = (0\ 0\ 1\ 0\ 0\ 0\ 0\ 1)^T$, $\mathbf{u}_3 = (Q_1\mathbf{u}_2)_{\geq\theta_1} = (0\ 0\ 1\ 0\ 0\ 0\ 0\ 1)^T = \mathbf{u}_2$. Hence $\mathbf{u}_\infty = \mathbf{u}_2 = (0\ 0\ 1\ 0\ 0\ 0\ 0\ 1)^T$ which coincides with $\mathbf{u}^{(J_\infty)}$ encoding $J_\infty = \{\texttt{c}, \texttt{nd}\}$ for DB_1^d that corresponds to the least 3-valued model $(\{\texttt{c}\}, \{\texttt{d}\})$ of comp(DB_0).

The procedure in Figure 6 shows that it is possible to compute the least model semantics purely within vector spaces, but unfortunately, it takes $O((2N)^3)$ time if implemented naively using matrix multiplication. On the other hand, although we do not explain algorithmic details here, it is possible to compute the least model of Horn clause programs by matrix operations in O(size(|DB|)) time by an elaborated implementation where size(DB) is the number of atoms occurring in DB.

4 SUPPORTED MODEL COMPUTATION

Suppose a program $\mathrm{DB} = \{a_1 \leftarrow B_i, \ldots, a_N \leftarrow B_N\}$ in $\mathcal{A} = \{a_1, \ldots, a_N\}$ is given and the least 3-valued model $I_\infty = (P_\infty, N_\infty)$ of comp(DB) has been computed. Put $U_\infty = \mathcal{A} \setminus (P_\infty \cup N_\infty)$. Atoms a in U_∞ cause infinite computation, or looping computation when an SLD-derivation is applied to $\leftarrow a$. As stated before, every supported model can be obtained by appropriately assigning **t** or **f** to atoms in U_∞. We construct a

[6] We identify 1 with **t** and 0 with **f** where the context is clear.

[7] $J_1 = \{a \mid a \leftarrow \texttt{true} \in \mathrm{DB}^d\} \cup \{na \mid na \leftarrow \texttt{true} \in \mathrm{DB}^d\}$. So $\mathbf{u}_1$ encodes unit clauses in DB^d.

supported model of DB by a divide-and-conquer approach. First we analyze U_∞ and detect *strongly connected components* (SCCs) in U_∞ explained next. We then process SCCs one by one, i.e. assign, from bottom SCCs upwards, **t** or **f** to atoms in each SCC to locally construct a supported model of the SCC. When this process is completed without failure, we have a supported model of DB.

In general, atoms in U_∞ have caller-caller dependency specified by DB and we express it as a *dependency graph* G_{U_∞} s.t. nodes are atoms in U_∞ and there is a directed edge from a node a_i to a node a_j iff there is a clause $a_i \leftarrow B_i$ is in DB and B_i contains a_j. An SCC is a set of atoms written as $[a] = \{a\} \cup \{b \mid$ there exist paths from a to b and from b to a in $G_{U_\infty}\}$ for some $a \in U_\infty$. Intuitively atoms in an SCC are those calling one another, directly or indirectly.

Note that SCCs in G_{U_∞} have a natural partial ordering. Let $[a]$ and $[b]$ be two SCCs. If there is a path from a to b but not from b to a, we write $[b] \prec [a]$. "$\prec$" is a partial ordering on SCCs. Using "$\prec$", SCCs are reverse-topologically sorted like $[S_1, S_2 \ldots]$ s.t. whenever $S_i \prec S_j$, $i < j$ holds. We can obtain this reverse-topologically sorted list by Tarjan's algorithm in $O(|V|+|E|)$ time where $|V|$ is the number of nodes, $|E|$ the number of edges of G_{U_∞}.

Look at DB_2 in Figure 7 which is a slight variant of DB_1. The least 3-valued model of comp(DB_2) is $(\emptyset, \emptyset)$ and all atoms are undefined, i.e. $U_\infty = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$. There are two SCCs, $[\mathtt{a}] = [\mathtt{b}] = \{\mathtt{a},\mathtt{b}\}$ and $[\mathtt{c}] = [\mathtt{d}] = \{\mathtt{c},\mathtt{d}\}$, and $[\mathtt{c}] \prec [\mathtt{a}]$ holds. They are reverse-topologically sorted into a list $SCC_{DB_2} = [\{\mathtt{c},\mathtt{d}\},\{\mathtt{a},\mathtt{b}\}]$.

$$DB_2 = \begin{cases} \mathtt{a} \leftarrow \neg \mathtt{b} \wedge \mathtt{c} \\ \mathtt{b} \leftarrow \neg \mathtt{a} \wedge \mathtt{c} \\ \mathtt{c} \leftarrow \neg \mathtt{d} \\ \mathtt{d} \leftarrow \neg \mathtt{c} \end{cases} \quad SCC_{DB_2} = [\{\mathtt{c},\mathtt{d}\},\{\mathtt{a},\mathtt{b}\}]$$

Figure 7: DB_2 having two SCCs.

Let $SCC_{DB} = [SCC_1, SCC_2, \ldots]$ be a reverse-topologically sorted list of SCCs in U_∞. We build a supported model of DB by processing those SCCs from left-to-right. Suppose SCC_js ($j < i$) preceding SCC_i have been processed and their atoms are already assigned **t** or **f**. Write $SCC_i = \{b_1, \ldots, b_K\}$ and let $DB_i^{SCC} = \{b_j \leftarrow W_j \mid 1 \le j \le K\}$ be a subprogram of DB about atoms in SCC_i. Since undefined atoms that appear in DB_i^{SCC} other than SCC_i are already assigned **t** or **f**, we construct a supported model of DB_i^{SCC} by appropriately assigning **t** or **f** to atoms in SCC_i when possible.

Our task w.r.t. SCC_i is to make every $b_j \Leftrightarrow W_j$ $(1 \le j \le K)$ in comp(SCC_i) true (in 2-valued logic) by determining the truth values of $b_1, \ldots, b_K$. Although this is easily formulated as a SAT problem and solvable by a SAT solver, we would like to exploit the form of iff completion which is lost in translation to CNF. Suppose $\mathrm{iff}(b) = b \Leftrightarrow c \wedge \neg d$ is in DB_i^{SCC} and $\{b, c, d\}$ are all undefined. We have to find their truth values that make iff(b) true. Basically this is an exhaustive search but since iff(b) is an equivalence formula, the truth value of the atom b, once determined, propagates to the body atoms on the right-hand side. For example, if **t** is assigned to b, $c = \mathbf{t}$ and $d = \mathbf{f}$ necessarily follow for iff(b) to be true. Or if **f** is assigned to b, we nondeterministically choose one of $\{c, \neg d\}$ and make the chosen literal false. Similarly for the disjunctive case such as $\mathrm{iff}(b) = b \Leftrightarrow c \vee \neg d$, an assignment $b = \mathbf{f}$ propagates to $c = \mathbf{f}$ and $d = \mathbf{t}$ in the body and so on. In this way, we make use of the iff completion to efficiently propagate truth values from one atom to another in SCC_i.

We conclude this section with our search procedure for supported models in Figure 8. Suppose we are given a program DB in a set of propositional atoms $\mathcal{A}$.

Step 1: Compute the least 3-valued model $I_\infty = (P_\infty, N_\infty)$ of comp(DB) via dualized program DB^d using the procedure in Figure 6 and extract undefined atoms $U_\infty = \mathcal{A} \setminus (P_\infty \cup N_\infty)$.

Step 2: Analyze U_∞ and obtain a reverse-topologically sorted list $SCC_{DB} = \{SCC_1, \ldots, SCC_M\}$ of SCCs in U_∞.

Step 3: For $i = 1$ to M, assign appropriately truth values $\{\mathbf{t}, \mathbf{f}\}$ to atoms in SCC_i so that the total assignment constitutes a supported model of DB, i.e. a 2-valued model of comp(DB).

Figure 8: Search procedure for a supported model of DB via dualized DB^d.

5 EXPERIMENT

In this section, we conduct an experiment with supported model computation[8] using the computation procedure described in Figure 8 which is implemented entirely by matrix operations provided by GNU Octave 4.2.2[9]. Our experiment is intended to

[8]The experiment is conducted on a PC with Intel(R) Core(TM) i7-8650U CPU (max 4GHz), 16 GB memory.

[9]For example, to implement **Step 2**, we represent a dependency graph G_{U_∞} by an adjacency matrix and implement Tarjan's algorithm on it to compute SCCs.

show the effectiveness of 3-valued model computation as a preprocessing step prior to 2-valued model computation.

Computing the least 3-valued model $I_\infty = (P_\infty, N_\infty)$ of comp(DB) at **Step 1** in Figure 8 has the effect of reducing the search space for supported model construction of DB by detecting atoms whose truth values are determined, or common to all supported models of DB. That is, since atoms in P_∞ (resp. atoms in N_∞) are true (resp. false) in any supported model of DB, we have only to consider truth value assignment for the remaining atoms, those in $U_\infty = \mathcal{A} \setminus (P_\infty \cup N_\infty)$ when searching for a supported model of DB.

We here conduct an experiment to measure the effect of **Step 1** on search space reduction. First we introduce *newly determined atom*s. They are atoms whose truth values are "newly determined" at **Step 1**. By "newly determined", we mean those atoms in $(P_\infty \cup N_\infty) \setminus (P_1 \cup N_1)$ because atoms in $P_1 \cup N_1$ are unit clauses (facts) or atoms having no clauses about them, and hence their truth values are immediately known. Since newly determined atoms are removed from the search space for supported model construction, we measure the effect of search space reduction by "reduction_rate" defined by reduction_rate = #newly_determined/n where #newly_determined is the number of newly determined atoms and n is the total number of atoms.

In this experiment, after setting the number of atoms $n = 100$ and a probability $p = 0.03$, we randomly generate a normal logic program $\mathrm{DB}_{(100,0.03)}$ in a set of propositional atoms $\mathcal{A} = \{a_1, \ldots, a_{100}\}$, compute its least 3-valued completion model and count #newly_determined_atom. When generating clauses in $\mathrm{DB}_{(100,0.03)}$, we specifically consider "base atoms" $\{a_1, \ldots, a_{10}\}$, and convert them to facts (unit clauses) $\{a_1 \leftarrow, \ldots, a_{10} \leftarrow\}$ or to tautologies $\{a_1 \leftarrow a_1, \ldots, a_{10} \leftarrow a_{10}\}$. When base atoms are converted to facts, $\mathrm{DB}_{(100,0.03)}$ will have more newly determined atoms than are converted to tautologies. To generate clauses $a \leftarrow B$ for the remaining atoms $\{a_{11}, \ldots, a_{100}\}$, we randomly pick up atoms in $\mathcal{A}$ with probability p, negate them with probability 0.5 and make a conjunction of resulting literals with probability 0.5 as the clause body B. Otherwise use their disjunction as B. Consequently, the body B contains average $n \cdot p = 100 \cdot 0.03 = 3$ atoms. We repeat this process 90 times and construct the remaining clauses about $\{a_{11}, \ldots, a_{100}\}$ in $\mathrm{DB}_{(100,0.03)}$.

Table. 1 depicts the statistics of this experiment (figures are average over 10 trials). There #empty_body denotes the average number of atoms having no clause about them in the randomly generated $\mathrm{DB}_{(100,0.03)}$. Their truth values are **f** in every supported model of $\mathrm{DB}_{(100,0.03)}$. #undef_atom is the average number of undefined atoms in the least 3-valued model of comp($\mathrm{DB}_{(100,0.03)}$). #newly_determined_atom is therefore computed as 100 − (#undef_atom + #zero_body + 10) when $\{a_1, \ldots, a_{10}\}$ are converted to facts. Otherwise it is 100 − (#undef_atom + #zero_body).

In the former case (see as_facts column), we observe #newly_determined_atom = 83.9 giving reduction_rate = 83.9%. It means, on average, 83.9% of atoms are newly assigned **t** or **f** at **Step 1** and removed from the search space for supported model construction. In the latter case (see as_tautology column) in which base atoms are converted to tautologies, we have more undefined atoms, resulting in a smaller #newly_determined_atom, 45.1 on average, but still 45.1% of atoms get their truth values automatically determined and removed from the search space at **Step 1**. Although the effect of preprocessing by computing the deterministic part of supported models at **Step 1** may vary depending on programs, as far as randomly generated programs in this experiment are concerned, we may say it greatly reduces the search space associated with supported model construction.

6 RELATED WORK

3-valued semantics of logic programs has been developed primarily from a theoretical perspective (Fitting, 1985; Kunen, 1987; Van Gelder et al., 1991; Naish, 2006; Barbosa et al., 2019). Fitting generalized the T_P operator associated with 2-valued logic programs P to a 3-valued operator Φ_P. He used Kleene's 3-valued logic and transfinite induction, which is similar to the inductive definition of $\{(P_n, N_n)\}$ given in Figure 1 where n is replaced by ordinals, and established the existence of the least 3-valued model of arbitrary normal logic programs (Fitting, 1985). However his semantics is highly undecidable, goes far beyond computable relations (at the cost of induction up to the Church-Kleene ordinal, i.e. the smallest non-recursive ordinal). Kunen later proposed to cut off Fitting's induction at ω and proved using a 3-valued ultra power model construction that the notion of $\mathrm{DB} \models_3 \phi$, a sentence ϕ being a 3-valued logical consequence

Table 1: The number of newly determined atoms.

base atoms $\{a_1, \ldots, a_{10}\}$	as_facts	as_tautology
#empty_body	3.6	12.4
#undef_atom	3.5	42.5
reduction_rate	83.9%	45.1%

of a program DB, is recursively enumerable (Kunen, 1987). Van Gelder proposed well-founded semantics (Van Gelder et al., 1991). The denotation of a program P, well-founded partial model, is defined as the least fixed point of some monotonic operator W_P associated with P, which gives a 3-valued model of comp(P). However, since W_P is asymmetric on the treatment of positive/negative occurrence of atoms in the clause body, well-founded semantics differs from our semantics; for example, p in a program $\{p \leftarrow p\}$ receives $\mathbf{f}$ in well-founded semantics.

Supported models of a program DB are 2-valued models of the completed program comp(DB) (Apt et al., 1988; Marek and V.S.Subrahmanian, 1992). They can represent solutions of a quite large class of combinatorial problems, and hence their efficient computation is of practical interest. Also it is well-known that stable models used in answer set programming (ASP) are a subclass of supported models and when propositional programs are finite and tight, they are identical (Erdem and Lifschitz, 2003). Our proposal to use 3-valued model computation as a preprocessing step to compute supported models looks new and is applicable to stable model computation as well. It eliminates, as the experiment in Section 5 shows, the extraneous need for finding the right assignment of $\{\mathbf{t},\mathbf{f}\}$ to the deterministic part of supported models. On the other hand, in ASP, stable models (or supported models) are computed by highly developed SAT technologies as in clingo (Gebser et al., 2019). It is an interesting future topic to merge our matricized approach with existing ASP computation mechanism.

7 CONCLUSION

We proposed to compute the least 3-valued completion model of a finite normal logic program DB in a vector space by first converting DB to an equivalent definite clause program DB^d, the dualized version of DB, and then computing its least 2-valued model in a vector space using a matrix representing DB^d, which is translated back to the least 3-valued completion model of DB. We then applied this 3-valued model computation to computing 2-valued completion models of DB, i.e. supported models of DB which are a super class of stable models. We constructed them by appropriately assigning $\mathbf{t}$ or $\mathbf{f}$ to the undefined atoms in the least 3-valued completion model of DB while guided by the completion form of clauses. We implemented the 2-valued and 3-valued completion model computation by matrix operations, and confirmed the effectiveness of 3-valued computation as a preprocessing step prior to 2-valued model computation. Assigning truth vales to undefined atoms found in this method is the next step to compute 2-valued supported models, and verification of efficiency of this part will be reported in a full version of this paper.

REFERENCES

Apt, K. R., Blair, H. A., and Walker, A. (1988). Foundations of deductive databases and logic programming. chapter Towards a Theory of Declarative Knowledge, pages 89–148.

Barbosa, J., Florido, M., and Costa, V. S. (2019). A three-valued semantics for typed logic programming. In *Proceedings 35th International Conference on Logic Programming, ICLP 2019 Technical Communications*, pages 36–51.

Erdem, E. and Lifschitz, V. (2003). Tight Logic Programs. *Theory and Practice of Logic Programming (TPLP)*, 3(4–5):499–518.

Fitting, M. (1985). A Kripke-Kleene semactics for logic programs. *Journal of Logic Programming*, 2:295–312.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2019). Multi-shot ASP solving with clingo. *TPLP*, 19(1):27–82.

Kunen, K. (1987). Negation in logic programming. *Journal of Logic Programming*, 4:289–308.

Marek, W. and V.S.Subrahmanian (1992). The relationship between stable, supported, default and autoepistemic semantics for general logic programs. *Theoretical Computer Science*, 103(2):365–386.

Naish, L. (2006). A three-valued semantics for logic programmers. *Theory and Practice of Logic Programming (TPLP)*, 6(5):509–538.

Sakama, C., Inoue, K., and Sato, T. (2017). Linear Algebraic Characterization of Logic Programs. In *Proceedings of the 10th International Conference on Knowledge Science, Engineering and Management (KSEM2017), LNAI 10412, Springer-Verlag*, pages 520–533.

Sato, T. (1990). Completed logic programs and their consistency. *Journal of Logic Programming*, 9:33–44.

Sato, T. (2017). A linear algebraic approach to Datalog evaluation. *Theory and Practice of Logic Programming (TPLP)*, 17(3):244–265.

Sato, T., Inoue, K., and Sakama, C. (2018). Abducing relations in continuous spaces. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI-ECAI-18)*, pages 1956–1962.

Van Gelder, A., Ross, K., and Schlipf, J. (1991). The well-founded semantics for general logic programs. *The journal of ACM (JACM)*, 38(3):620–650.