Model Expansion in the Presence of Function Symbols Using Constraint Programming

Broes De Cat
Bart Bogaerts
Jo Devriendt
Marc Denecker

Report CW 644, July 2013

Department of Computer Science, KU Leuven

Abstract

The traditional approach to model expansion (generating models of a logic theory extending a partial structure) is to reduce the theory to a propositional language and apply a search algorithm to the resulting theory. Function symbols are typically replaced by predicate symbols representing the graph of the function, an operation that blows up the reduced theory.

In this paper, we present an improved approach to handle function symbols in a ground-and-solve methodology, building on ideas from Constraint Programming. We do so in the context of $\text{FO}(\cdot)^{\text{IDP}}$, the knowledge representation language that extends First-Order logic with, among others, inductive definitions, arithmetic and aggregates. A model expansion algorithm is developed, consisting of (i) a grounding algorithm for $\text{FO}(\cdot)^{\text{IDP}}$ that is parametrized by the function symbols that are allowed to occur in the reduced theory, and (ii) a search algorithm for unrestricted, ground $\text{FO}(\cdot)^{\text{IDP}}$. The ideas are implemented within the IDP knowledge-base system and experimental evaluation shows that both more compact groundings and improved search performance are obtained.

Keywords: Extended First-Order Logic, Declarative Modeling, Constraint Programming.

CR Subject Classification: F.4.1, I.2.4.
Model Expansion in the Presence of Function Symbols Using Constraint Programming

Broes De Cat, Bart Bogaerts, Jo Devriendt and Marc Denecker
Department of Computer Science, KU Leuven, Belgium
Email: firstname.lastname@cs.kuleuven.be

Abstract—The traditional approach to model expansion (generating models of a logic theory extending a partial structure) is to reduce the theory to a propositional language and apply a search algorithm to the resulting theory. Function symbols are typically replaced by predicate symbols representing the graph of the function, an operation that blows up the reduced theory.

In this paper, we present an improved approach to handle function symbols in a ground-and-solve methodology, building on ideas from Constraint Programming. We do so in the context of FO(·)IDP, the knowledge representation language that extends First-Order Logic (FO) with, among others, inductive definitions, arithmetic and aggregates. A model expansion algorithm is developed, consisting of (i) a grounding algorithm for FO(·)IDP that is parametrized by the function symbols the are allowed to occur in the reduced theory, and (ii) a search algorithm for unrestricted, ground FO(·)IDP. The ideas are implemented within the IDP knowledge-base system and experimental evaluation shows that both more compact groundings and improved search performance are obtained.

I. INTRODUCTION

Model generation is a widely used problem solving paradigm. A problem is specified as a theory in a declarative logic in such a way that models of the theory represent solutions to the problem. A closely related paradigm is bounded Model Expansion (MX). Here, a partial input structure over a finite and known domain is extended into a complete structure that is a model of a given theory. These paradigms are studied and applied in the fields of Constraint Programming (CP) [1], Answer Set Programming (ASP) [26] and Knowledge Representation (KR) [3].

A state-of-the-art approach is to reduce the input theory, formulated in an expressive logic, in a model-equivalence preserving way to a theory in a fragment of the language supported by some search algorithm. Afterwards, this algorithm searches for models of the theory. For example, model generation/ expansion for the language FO(·) [9] is performed by reducing theories to a ground fragment of FO(·) for which a search algorithm is available. The term grounding refers to both the reduction process and to its outcome; the 2-step approach is called ground-and-solve.

A first generation of MX systems used search algorithms for (pseudo)-propositional languages, such as Clausal Normal Form (SAT solvers) and ground ASP (ASP solvers). An important bottleneck of such systems is the blowup caused by grounding the input theory, as the size of the theory increases rapidly with the size of the domain and the nesting depth of quantified variables. One apparent approach to reduce the nesting depth of quantified variables is to replace predicate symbols with function symbols wherever possible, as follows.

Example 1.1. Consider the well-known 2-D packing-problem for squares: given a set of squares with known size and a rectangular area of known size, position all squares in a non-overlapping fashion within the area (if possible). One of the constraints, that two squares should not overlap horizontally, can be expressed as follows, using pos$_s(id, x)$ to express that the left side square $id$ is at $x$ and sz$(id)$ for the size of $id$:

$$\forall id_1 \ id_2 \ x_1 \ x_2 : id_1 \neq id_2 \land pos_s(id_1, x_1) \land pos_s(id_2, x_2) \Rightarrow (x_1 + sz(id_1) \leq x_2 \lor x_2 + sz(id_2) \leq x_1).$$

In fact, pos$_s$ represents a function mapping squares to x-coordinates, so it can be rewritten using a function $f_s(id)$ for $x$:

$$\forall id_1 \ id_2 : id_1 \neq id_2 \Rightarrow (f_s(id_1) + sz(id_1) \leq f_s(id_2) \lor f_s(id_2) + sz(id_2) \leq f_s(id_1)).$$

Next to being a more natural way to express the constraint, the rewriting halves the quantifier depth. However, if the target solver only takes propositional input, the function symbols are eliminated again during the reduction phase, replacing function symbols by predicate symbols and adding additional quantifiers. In fact, in the example, it comes down to transforming the latter sentence into the former one.

Recently, research is being done in ASP to incorporate techniques from CP, giving rise to the field of ASP modulo CSP (CASP) [27]. In CASP, the ASP language is extended with constraint atoms, atoms that stand for the constraints of a CSP problem [19][14], and can, for example, contain function symbols. Second, search algorithms have been developed that allow ground constraint atoms (instead of only propositional atoms) in the input. This gives rise to more compact groundings that often also yield better propagation. Among those next generation systems are the systems Clingcon[27], EZ/CSP[2], Mingo[20] and Incal[12].

In this paper, we work in the context of the language FO(·)IDP, the language of the knowledge-base system (KBS) IDP [18]. FO(·)IDP extends FO with, among others, inductive definitions, aggregates and arithmetic. We show that for FO(·)IDP, allowing the grounding to contain function terms in fact produces a general form of such “constraint atoms”, without extending the language. In the above example, $f_s(id_1) + sz(id_1) \leq f_s(id_2)$ is such an atom, for which efficient propagation techniques exist in the field of CP. We present a model expansion algorithm for FO(·)IDP that exploits...
this idea. It of (i) an algorithm to ground FO(·)IDP theories without eliminating all function symbols from the grounding and (ii) a search algorithm for general, ground FO(·)IDP. As different search algorithms often support different sets of function symbols, the grounding algorithm is parametrized by the set of functions allowed to occur in the grounding. The search algorithm, extends the search algorithm of the state-of-the-art solver MINISAT(ID) [21] using the technique of lazy clause generation[30], an approach to support finite-domain constraints in a SAT-solver by encoding propagation as clauses (for details, see Section IV). The algorithms are implemented within IDP (i) and as the solver CONSTRAINT-IDP (ii).

We take terminology from the logic-based point of view to model generation. Below, we provide a short overview of coinciding notions from CP and ASP. A theory \( T \) can be seen as a set of constraints (CP) or a logic program (ASP). Symbols are by default non-defined \( t \) uninterpreted; constants (0-ary functions symbols) coincide with variables in the Constraint Programming sense (CP-variables) and \( n \)-ary \((n > 0)\) function symbols can be seen as \( n \)-dimensional arrays of CP-variables.

A (partial) interpretation coincides with a (partial) assignment to CP-variables; a model of \( T \) is a total interpretation satisfying \( T \), i.e., a solution (CP) or answer set (ASP). A domain is a set of domain elements, e.g., the set of values a CP-variable might take. Following CP-terminology, the domain of a function symbol refers to the set of values it can map to. A variable is a placeholder for instantiation with domain elements.

The paper is organized as follows. In Section II, the language FO(·)IDP is introduced. Next, the algorithms for grounding and search are presented in Section III, respectively Section IV. Experimental evaluation is presented in Section V; related work and concluding remarks in Section VI.

II. PRELIMINARIES

We assume familiarity with FO. The notation FO(·) [9] denotes the family of extensions of FO with new language constructs. The language we consider in this paper is the language FO(·)IDP. It is a many-sorted extension of FO with aggregate functions, arithmetic and inductive definitions. We know give an overview of the language.

A vocabulary \( \Sigma \) consists of a set \( \Sigma_n \) of types denoted \( \tau \) and a set \( \Sigma_\tau \) of typed predicate symbols denoted \( P, Q, R \) and function symbols denoted \( f, g, h \). For each type \( \tau \), \( \Sigma \) includes a unary predicate symbol \( T(\tau) \) representing all elements in \( \tau \).

Variables \( x, y \), terms \( t \), atoms \( A \), literals \( L \), domain elements \( d \), and FO-formulas \( \varphi \) are defined as usual. A domain atom(domain term) is an atom(term) consisting of a predicate(function) symbol applied to a tuple of domain elements.

We use \( e \) to denote domain terms and \( e \) to denote domain elements or domain terms. The set of symbols of a theory \( T \) is denoted vec(\( T \)). Given two tuples \( \pi \) and \( \pi' \) of terms of equal length \( n \), \( \pi = \pi' \) denotes the conjunction \( x_1 = x'_1 \land \ldots \land x_n = x'_n \). A term \( t \) containing occurrences of a term \( t' \) is denoted as \( t[t'/t] \); the replacement of \( t' \) in \( t \) by \( t'' \) is denoted as \( t[t'/t''/t] \) (similarly for formulas).

An interpretation for a type \( \tau \) is a set of domain elements \( D_\tau \). A (partial) interpretation for a predicate symbol \( P(\tau) \) consists of two disjoint subsets of \( D_{\tau_1} \times \cdots \times D_{\tau_n} \), denoted as \( P_\tau \) and \( P_{\tau.f} \). An (partial) interpretation for a function symbol \( f(\tau) : \tau' \) is a function mapping elements of \( D_{\tau_1} \times \cdots \times D_{\tau_n} \) to a non-empty subset of \( \tau' \). A (partial) \( \Sigma \)-interpretation \( I \) is then an interpretation for all symbols in \( \Sigma \); we use \( s' \) to refer to the interpretation of a symbol \( s \) in \( I \). An atom \( P(\bar{d}) \) is true in \( I \) if \( P(\bar{d}) \in I^P \); false if \( P(\bar{d}) \in I^P_{\bar{d}} \) and unknown otherwise. An atom \( f(\bar{d}) = d' \) is true in \( I \) if \( \{d'\} = f(\bar{d}) \), unknown if \( \{d'\} \subseteq f(\bar{d}) \) and false otherwise. An interpretation for a predicate symbol \( P \) is two-valued if \( P_\tau = P_{\tau.f} \); an interpretation for a function symbol \( f \) is two-valued if all images are singletons. An interpretation is two-valued if the interpretation of all its symbols are two-valued. For a two-valued interpretation \( I \), \( \varphi(\bar{x}) \) denotes the value of a formula \( \varphi \) (a term \( t \)) under \( I \) as usual. A well-typed expression is one in which the type of each argument matches with the type of its argument position. Badly typed atoms are false. We say a term \( t \) (formula \( \varphi \)) is ground-evaluatable in \( I \) if all symbols in \( t \) are two-valued. In this paper, we only consider interpretations where all types are finitely interpreted and with a total order on all domain elements; in that case, define \( \text{min}_{\tau} \) as the function mapping to the smallest element of \( \tau \) and \( \text{pred}_{\tau} \) as the (partial) predecessor function over \( \tau \).

We extend the notion of term to include aggregate terms. A set expression is of the form \( \{\tau : \varphi : t \} \) (if there are no local variables) or a union of set expressions \( \{\{\tau_1 : \varphi_1 : t_1\} \cup \{\tau_2 : \varphi_2 : t_2\} \) (denoted shortly as \( \{\{\tau_1 : \varphi_1 : t_1, \tau_2 : \varphi_2 : t_2\} \) ). Given an interpretation \( I \) and an assignment \( \pi \) to the free variables \( \pi \) of the set expression, the interpretation \( \{\pi : \varphi[\pi]/d \} : t[\pi]/d \} \) is the multiset \( \{t[\pi]/d, t[\pi]/d \} \). Thus, in the context of a given assignment for the variables \( \pi \), the expression denotes the multiset of tuples \( t \) for which \( \varphi \) holds. Aggregate terms are of the form \( \text{agg}(S) \), with \( \text{agg} \) an aggregate function (cardinality, sum, product, minimum or maximum) and \( S \) a set expression. The cardinality function then maps a set interpretation to the number of elements in the set. The aggregate functions sum, product, minimum and maximum map a set to respectively the sum, product, minimum and maximum of the elements in the set, or to 0, respectively 1, +∞ and −∞ if the set is empty.

Aggregate terms can occur nested in other aggregates; in this paper however, aggregate terms occurring in a definition cannot contain any symbols defined in that definition.²

²In fact, the results in this paper are also correct for such aggregate terms if they do not occur nested in other aggregate terms.
of a definitional rule is false, its head cannot be derived and is false (unless another rule derives it). This intuition coincides exactly with inductive definitions as in mathematical texts.

The completion of $\Delta$ for a symbol $P$, defined in $\Delta$ by the rules $\forall T_i : P(T_i) \leftarrow \phi_i$ with $i \in [1, n]$, is the set consisting of the sentence $\forall \tau : \exists \phi \Rightarrow P(\tau)$ for each $i \in [1, n]$ and the sentence $\forall \tau : P(\tau) \Rightarrow \bigvee_{i \in [1, n]} (\tau = T_i \wedge \phi_i)$. This set is denoted as $comp_{\Delta}$, the union of all these sets for $\Delta$ as $comp_{\Delta}$. It is well-known that if $I \models \Delta$ then $I \models comp_{\Delta}$ but not vice-versa (e.g., the inductive definition expressing transitive closure is stronger than its completion).

For a vocabulary $\Sigma$ and a structure $I$ over $\Sigma$, two theories $T$ and $T'$ are $(\Sigma, I)$-equivalent if for each model $M$ of $T$ that extends $I$, its restriction to $\Sigma$ can be extended to a model of $T'$ extending $I$, and vice-versa and the extensions are unique.

A formula is in Negation Normal Form (NNF) if $\neg$ only occurs directly in front of atoms and if conjunctions and disjunctions are in left-associative form (e.g., $A \lor (B \lor \ldots)$). We assume, without loss of generality, that our sentences and rule bodies are NNF.

III. GROUNDING TO PARAMETRISED GROUND FO($\cdot$)

This section describes an algorithm to construct the grounding of a theory $T_n$ over $\Sigma$ in the context of a 3-valued, consistent interpretation $I_n$. The algorithm transforms $T_n$ to a $(\Sigma, I)$-equivalent ground — quantifier-free — theory $T_n$ and a “mapping” theory $T_m$ consisting of explicit definitions for symbols of $\Sigma$ that were eliminated from $T_n$.

The algorithm takes as parameter a set ResF of “residual” function symbols, function symbols allowed in $T_n$. In our algorithm, functions $f/n$ not in ResF are replaced by their “graph” predicate symbol $g f/n + 1$. If ResF is empty, then all atoms in the grounding will be domain atoms; by translating these into propositional symbols, such a theory can be mapped into an “equivalent” propositional theory.

The grounding process is described as two stratified sequences of $(\Sigma, I)$-equivalence preserving rewrite rules, rewriting the theories $T_g$ and $T_n$. Theory $T_g$ is initialized as $T_{\Delta n}, T_m$ as the empty set. The rewrite rules operate on $T_g$, substituting expressions or rules by simpler ones, and sometimes introducing new definitions to $T_g$ or $T_m$. E.g., $\neg \neg \phi \Rightarrow \phi$ is the rule that replaces occurrences of $\neg \neg \phi$ in $T_g$ by $\phi$.

A. Phase 1: simplifying the syntax

The first phase consists of iterated rewriting of $T_g$ by the rewrite rules specified below. The rewriting process terminates when no more rules are applicable.

- $\phi \equiv \psi \equiv \phi \equiv \psi \land \psi \Rightarrow \phi$.
- $\phi \Rightarrow \psi \Rightarrow \neg \neg \phi \lor \psi$.
- $\neg \neg \phi \Rightarrow \phi$.
- $\neg(\phi \lor \psi) \Rightarrow \neg \phi \land \neg \psi$.
- $\neg(\phi \land \psi) \Rightarrow \neg \phi \lor \neg \psi$.
- $\neg(\exists x : \phi) \Rightarrow \exists x : \neg \phi$.
- $\neg(\exists x : \phi) \Rightarrow \forall x : \neg \phi$.

- $(t \sim t') \Rightarrow t \neq t'$. We use $\sim$ to denote a comparison operator such as $\leq, <, =, \neq, \ldots$ and $\neq$ denotes respectively $>, \geq, =, \ldots$.

- $\phi \land \psi \land \gamma \Rightarrow \phi \land (\psi \land \gamma)$.
- $\phi \lor \psi \lor \gamma \Rightarrow \phi \lor (\psi \lor \gamma)$.

Unnest function terms $f(T) \Rightarrow A[f(T)] \Rightarrow \exists x : f(T) = x \land A[f(T)/x]$ where $A$ is an occurrence of an atom in an FO sentence or rule body and $A$ is not of the form $f(T) = t$. $A[f(T)] \leftarrow \phi \Rightarrow \forall y : A[f(T)/y] \leftarrow f(T) = y \land \phi$.

These rewrite rules eliminate $\equiv$ and $\Rightarrow$, drive negation deeper and flatten conjunctions and disjunctions. In the resulting theory, negation is in front of atoms of user-defined symbols. All occurrences of function symbols $f \notin ResF$ are top left symbols in equalities $f(T) = t$. Note, if ResF is empty, such atoms are of the form $f(T) = t$ with $t_1, \ldots, t_n.t$ either domain elements (e.g., natural numbers) or variables. As final step in this phase, function symbols are replaced by their graph as follows. For each function symbol $f/n \notin ResF$, we introduce a new predicate symbol $g f/n + 1$. apply the rewrite rule $f(T) = t \Rightarrow g f(T), t$ add $\forall \tau : \phi(g f(T), y : 1) = t$ to $T_g$ and add $\forall \tau : f(T) = t$ to $T_m$.
introduced constant over the type of \( t \). Additionally, \( t = c_i \) is added to \( T_g \). The rule is not applied if \( t \) is a domain element or occurs in an atom of the form \( P(\overline{\tau}) \), \( f(\overline{\tau}) \sim e_0 \) or \( \text{agg}(\{L_1 : e_1\} \cup \ldots \cup \{L_n : e_n\}) \sim e_0 \).

- **Simplify**
  
  \[ \neg t \mapsto f \quad \neg f \mapsto t \]
  
  \[ \psi \lor t \mapsto t \quad \psi \land f \mapsto f \]
  
  \[ \psi \lor f \mapsto \psi \quad \psi \land t \mapsto \psi \]
  
  \[
  \forall \tau \in D : t \mapsto t \quad \exists \tau \in D : t \mapsto \tau \\
  \forall \tau \in D : f \mapsto f \\
  \forall \tau \in \emptyset : \psi \mapsto f \\
  \{ \tau \in \emptyset : \psi : t \} \mapsto \{ \tau \in D : f : t \} \mapsto \{ f : 0 \}
  \]

After application of the above rewrite rules, we obtain a theory in Ground Normal Form (GNF).

**Definition III.1.** An FO(\( \cdot \)) theory \( T \) is in *Ground Normal Form* (GNF) if all its sentences and rules are of one the following forms (with all \( L_i \)'s domain literals):

\[
\begin{align*}
L_1 \lor \ldots \lor L_n, & \quad Q(\overline{\tau}) \quad f(\overline{\tau}) \sim e_0, \\
\text{agg}(\{L_1 : e_1\} \cup \ldots \cup \{L_n : e_n\}) & \sim e_0, \\
P(\overline{\tau}) \leftarrow L_1 \land \ldots \land L_n, & \\
P(\overline{\tau}) \leftarrow L_1 \lor \ldots \lor L_n, \\
P(\overline{\tau}) \twoheadleftarrow Q(\overline{\tau}), & \\
P(\overline{\tau}) \leftarrow f(\overline{\tau}) \sim e_0, \\
P(\overline{\tau}) \leftarrow \text{agg}(\{L_1 : e_1\} \cup \ldots \cup \{L_n : e_n\}) \sim e_0.
\end{align*}
\]

**Theorem III.2.** For input \( T_{in}, T_g \) and \( \text{ResF} \), let \( T_g \) be the computed theories at any time during the rewrite process. Then \( T_{in} \) and \( T_g \cup \text{ResF} \) are \( \{\Sigma, \Xi\} \)-equivalent. The rewrite-process terminates and the resulting theory \( T_g \) is in GNF and contains only function symbols in \( \text{ResF} \).

The equivalence follows from the fact that each rewrite rule preserves \( \{\Sigma, \Xi\} \)-equivalence. That the resulting theory is in GNF follows from the fact that none of the rewrite rules apply in the context of a GNF theory and that at least one rewrite rule is applicable to any theory not in GNF.

Termination of phase 1 is straightforward. To prove termination of phase 2, it can be shown that a well-founded order exists on theories for the presented rewrite rules. This order depends among others on the nesting depth of symbols, the nesting and domain size of quantifications and the number of occurrences of symbols in the theory. The formal presentation of the well-founded order is out of the scope of this paper.

**C. Concrete grounding algorithm**

The rewrite process of the previous section is not confluent. By imposing different rewrite strategies, it can be instantiated to a class of sound grounding algorithms. To obtain a state-of-the-art grounding algorithm, one should select an instantiation that minimizes the number of traversals through formulas in search for applicable rewrite rules, the memory and time complexity of the algorithm, the grounding size, etc.

The rewrite strategy that is implemented in our system is quite complex and a full presentation is out of the scope of this paper; we highlight the most important considerations here:

- **Simplify** is performed top-down and depth-first. This allows to simplify formulas early and reduces the memory overhead of storing partial results.
- **Simplify** and **Evaluate** are applied eagerly, as they may considerably reduce the size of formulas.
- The number of introduced symbols should be minimized. E.g., it is important to avoid the creation of multiple Tseitin symbols for multiple occurrences of the same term, atom or formula.

An important optimisation is to first make \( T_{in} \) more precise by applying symbolic propagation for \( T_{in} \) to it. This leads to a more precise 3-valued structure \( T_{in}' \) that approximates all instances of \( T_{in} \) that are models of \( T_{in} \). This process was described first in [34] and later for the restriction to FO in [31] where it was called lifted unit propagation, as this propagation is indeed a symbolic version of unit propagation. With the refined structure \( T_{in}' \), the ground theories are sometimes orders of magnitude smaller than w.r.t. \( T_{in} \) [35].

**IV. MODEL EXPANSION FOR GENERAL HAND FOO(\( \cdot \))**

In this section, we present an MX algorithm which takes as input a general ground FO(\( \cdot \)) theory \( T_g \) in GNF and a 3-valued input structure \( T_{in} \), as before, we assume that all types \( \tau \) are interpreted as finite sets \( \tau^{\infty} \) of domain elements.

The algorithm is based on an existing MX algorithm for function-free GNF, implemented in the system MINISAT(1D) and described in [22][21]. That algorithm is a conflict-driven clause-learning (CDCL) search algorithm, extended to handle inductive definitions and aggregations. Recall that function-free GNF can be obtained by running the algorithm of the previous section with \( \text{ResF} = \emptyset \). The algorithm developed here uses a generalisation of the technique of lazy clause generation [30] to support full GNF, explained later in this section.

**A. Adapt existing CDCL algorithm to our setting**

The state of the algorithm consists of a theory \( T_v \), and a three-valued interpretation \( I \). We present \( I \) as the sequence of its true literals, ordered by the time at which the literals were derived. A literal \( L \), in this sequence is annotated \( L^D \) if it is a decision literal; other literals were derived by propagation. Initially, \( T_v \) is the input theory \( T_g \) and \( I \) is the empty set. For ease of presentation, we use a slight adaptation of GNF in the rest of the paper: any sentence \( A \), with \( A \) one of the atoms \( Q(\overline{\tau}), f(\overline{\tau}) \sim e_0 \) or \( \text{agg}(\{L_1 : e_1\} \cup \ldots \cup \{L_n : e_n\}) \sim e_0 \), is generalized as an equivalence \( P(\overline{\tau}) \sim A \). Any such sentence \( A \) in \( T_g \) is then added as the sentence \( t \sim A \) to \( T_v \).

As an initial step of the algorithm, definitions \( \Delta \) in \( T_v \) are simplified. If \( \Delta \) is not recursive (or if it can be stratified), it can be split in a set of subdefinitions \( \Delta_1, \ldots, \Delta_n \), as shown in [9]. These are added to \( T_v \) and \( \Delta \) is removed from it.

A number of inference rules operate on such states. The first four rules describe a basic CDCL SAT-solver: **Decide**: Select non-deterministically a domain literal \( L \) such that \( L^D = U \), and append \( L^D \) to \( I \). **UP**: Apply unit propagation to a
clause in $T_m$ and append the derived literal $L$ to $I$. 

**Fail:** If $I$ is inconsistent and contains no decision literals, the algorithm returns “unsatisfiable”. 

**Learn:** If $I$ is inconsistent and contains decision literals, conflict-driven clause-learning is applied to $I$ and $T_m$ to construct a learnt clause $C$ which is added to $T_m$. Backjumping to the level of the second youngest literal of this clause occurs. The output of the algorithm is either fail or a three-valued interpretation $I$ expanding $T_m$ such that every more precise two-valued interpretation $I'$ is a model of $T_m$.

The remaining propagation rules, presented in the next section, then serve to perform propagation on the non-clausal components of $T_m$. In the MINISAT(1D) algorithm, this consists of four additional rules, **Aggregate**, **Completion**, **Unfounded** and **Wellfounded**. The first checks for propagation over aggregate expressions by reasoning on the bounds of the aggregate function (the minimum and maximum value the function can take still can take in a partial structure). The latter two rules apply to inductive definitions. The rule **Completion** is only executed in the initial phase; it applies to a definition $\Delta$ and adds its completion to $T_m$. If $\Delta$ is equivalent with its completion (for example for Tseitin symbols introduced only for unification purposes), the algorithm can be dropped from $T_m$, as shown in [9].

**Unfounded** searches for unfounded sets[33] in a definition $\Delta$ and if an unfounded set $U$ is found, propagates all its atoms as $f$ (i.e., it appends $\sim U$ to $I$). When $I$ is a 2-valued interpretation, **Wellfounded** checks if $I$ is a well-founded model of a definition $\Delta$, as shown in [32]. In what follows, these rules will be extended (and new ones will be added), to handle the more general format of GNF.

We omit a discussion on CDCL improvements such as the 2-watched literal scheme and restarts; they can be incorporated straightforwardly in the presented algorithm. The experimental evaluation is based on a state-of-the-art CDCL algorithm.

**B. Approach to extend to GNF**

Any GNF theory can be transformed into a $\{\Sigma, I\}$-equivalent function-free GNF theory. In that case, the inference rules presented above are sufficient for a complete algorithm. One approach to obtain such a theory was already presented in Section III: to apply the rewrite algorithm with an empty set $\mathbb{E}$. However, instead of generating such a function-free theory eagerly, before search, in the rest of the section we present a concrete algorithm to generate such a function-free theory lazily (i.e., during search). The algorithm is based on the technique of **lazy clause generation**, presented in [30]. Lazy clause generation alleviates the blowup of creating the full function-free ground theory in advance in two ways: first it uses smarter technique than graphing functions, and second, it only generates these clauses when they would contribute to the search, i.e. on the moment that they would propagate. We generalise the scheme by not only lazily generating clauses but lazily generating GNF sentences. To avoid infinite loops, we impose the following partial order on GNF expressions; only smaller expressions can be generated (w.r.t. this ordering).

- **definitional rules** $\Leftarrow$ **reified aggregates** $\Leftarrow$ **P** $\Leftarrow$ **Q(π)** $\Leftarrow$ **P** $\Leftarrow$ $f(\bar{π}) \sim e_0$ $\Leftarrow$ **P** $\Leftarrow$ $f(\bar{d}) \sim e_0$ $\Leftarrow$ **clauses**

We assume $\bar{π}$ contains at least one domain terms (distinguishing it from the case $P \Leftarrow f(\bar{d}) \sim e_0$); additionally, we assume expressions over the operator $\leq$ are ordered lower below similar expressions over other comparison operators.

In the rest of the section, we show how the various GNF expressions that possibly contain function terms are supported. For each of these, the presentation consists of three components. First, a set of (non-ground) sentences of the form $\forall \bar{x} : \varphi \Rightarrow L_i$; intuitively, these will be the set of propagations or decompositions we consider for the expression at hand, propagating the right-hand side (the head) when the left-hand side (the body) is true. Second, a discussion on how to quickly find instances of $\mathbb{T}$ for which $\varphi$ holds in $I$ and $L_i$ is not true. The algorithm then consists of adding the sentence $\varphi[\bar{x}/\bar{d}] \Rightarrow L_i[\bar{x}/\bar{d}]$ for the relevant instantiations $\bar{d}$ of $\mathbb{T}$. Third, a discussion on when such derived sentences will be added to $T_m$, which will depend on the expression at hand. As discussed previously, the type of the derived sentences should be ordered below the type of the original expression.

**C. Encoding functions**

To handle constraints on functions $f/n$ with domain $D$ in a solver that decides on domain atoms, we use the range encoding[30]. A domain term $t = f(\bar{d})$ with $f$ a function mapping to the domain $D = \{d_1, \ldots, d_n\}$ is encoded as the set of propositional symbols $\{T_{i \leq d_1}, \ldots, T_{i \leq d_n}\}$. For each $t$, we define $\min_t = \max_{d \in D} \{d \mid T_{i \leq d} = f\}$ and $\max_t = \min_{d \in D} \{d \mid T_{i \leq d} = 1\}$. The range of $t$ is then defined as $[\min_t, \max_t]$. The values $\min_t$ and $\max_t$ can be computed from $T$, but an efficient algorithm should store them and adapt them incrementally whenever $I$ changes.

The range encoding is selected over encoding the function as a set of equalities $T_{i = d}$, as the encoding of inequalities is more compact (and the encoding for equalities is only a factor $2$ larger) and choices on encoding atoms more often eliminate subsets of the domain instead of just one value. A more in-depth comparison is provided in [30].

In the sequel, for domain term $t$ and value $d$, we use $[t \leq d]$ to denote the atom $T_{i \leq d}$ if $d \in D = \{d_1, \ldots, d_n\}$, the atom $t$ if $d < d_1$, the atom $t'$ if $d > d_n$, and otherwise the atom $T_{i < d}$, with $d'$ the smallest domain element in $D$ larger than $d$. All other comparison operators $\sim$ can be defined in terms of $\leq$. We use $[t \sim d]$ as a shorthand for those rewritings. E.g., $[t \neq d]$ denotes $[t \leq d_{i-1}] \lor \sim[t \leq d_i]$.

The following set of non-ground clauses represents the dependencies between those symbols.

- $[t \leq d_i]$  
  $\forall x \in D - d_i : [t \leq x] \Rightarrow [t \leq next(x)]$  
  $\forall x \in D - d_i : [t > x] \Rightarrow [t > prev(x)]$

The propagation rule **Encode** is applied to a domain term $t$ the first time it appears in $T_m$, and it adds the grounding of the above formulas to $T_m$. For small domains $D (|D| < 100)$, this is done eagerly; for larger ones this is done lazily as described in [30]. We do not elaborate the details here. Additionally, to take care of interpreting $f$ when we have a model of the encoding clauses, **Encode** adds the mapping sentence $\forall x \in D : [f(\bar{d}) = x] \Rightarrow f(\bar{d}) = x$ to $T_m$.

**Example IV.1.** Consider the theory $T_0$ consisting only of the sentence $P \Leftarrow f(1) \leq 3$, with $f$ typed as $f(\tau) : \tau'$. 


\[ \tau \text{ interpreted as } D, \tau \text{ as } D'. \text{ Encode will then add the ground-}
\]n ing of the above sentences for \( t = f(1) \). It does not add instantiation for any other term \( f(d), d \neq 1 \), which has an important impact if \( D \) is large. In this case, the result of MX is a three-valued interpretation of which any two-valued extension is a model of the theory. For example, interpretation \( I = \{ P, f(1) = 3 \} \) contains enough information: all structures more precise than \( I \) are models of \( T_g \).

### D. Comparison constraint

The propagation rule \textbf{Comparison} applies to constraints \( P \iff c \leq c' \), with \( P \) a domain atom and \( c \) and \( c' \) domain terms over different domains \( D \) and \( D' \). The propagations we consider can be represented as the following sentences.

\[
\forall x \in D \cup D': \begin{cases}
    c \leq x \land [c' \geq x] & \Rightarrow P; \\
    c > x \land [c' < x] & \Rightarrow \neg P.
\end{cases}
\]

It is easy to see that together with the encoding of \( c \) and \( c' \), this set of sentences is \( S \cup I \)-equivalent to the original constraint. Comparison constraints over comparison operators other than \( \leq \) are converted into 1 or 2 comparison constraints over \( \leq \) (with Tseitin introduction in the latter case).

Instantiations are generated as follows. \textbf{Comparison} checks for each of the non-ground sentences whether the body is true, but only for instantiations of \( x \) with \( \min c, \max c, \min c' \) and \( \max c' \). This is checked whenever one of those values increases (for min) or decreases (for max) and whenever \( P \) becomes assigned. It is straightforward to show that this is sufficient, i.e., when \( \text{UP, Encode and Comparison} \) are at fixpoint (without conflict), none of the above sentences has a true body and an unknown or false head for any instantiation.

#### Example IV.2

Consider a constraint \( P \iff c \leq c' \), with \( c \) a range of \([3, 10]\), \( c' \) a range of \([7, 20]\) and \( P \) true in \( I \). When \( I \) is extended with \([c \geq 8]\) to \( I' \), \textbf{Comparison} checks for \( x = 8 \) which of the left-hand sides are true, which is the case for the sentence \([c \geq x] \land P \Rightarrow [c' \geq x] \). As the head is not true in \( I' \), the sentence is added to \( T_g \) (in clausal form) and \textbf{UP} will derive \([c' \geq 8]\).

### E. Aggregates

Next, we introduce propagation rules for sentences of the form \( P \iff \text{agg}([L_1 : e_1] \cup \cdots \cup [L_n : e_n]) \leq e_0 \) where \text{agg} is either a maximum or sum aggregate function. As above, other comparison operators can be rewritten into constraints over \( \leq \). Cardinality constraints are rewritten straightforwardly into sum constraints and minimum into maximum constraints. The rules for product aggregates are not presented here, as they are similar to those for sum (although complicated by the non-monotonicity of product for terms with negative values).

The rule \textbf{Encode} \(_{\text{max}} \) rewrites a maximum constraint \( P \iff \max(S) \leq c \) into the following sentences

\[ P \land L_i \Rightarrow e_i \leq e_0 \quad \text{for each } i \in [1, n] \]

\[ \neg P \Rightarrow \bigvee_{i \in [1, n]} (L_i \land c_i > e_0) \]

As the rewriting consists of only \( n + 1 \) ground sentences, it is done eagerly for any maximum aggregate constraint in \( T_g \).

Enumerating the clauses generated from a sum constraint, by the \textbf{Encode} \(_{\text{sum}} \) propagation rule, is out of the scope of this paper, we only give an example: the sentence

\[ \left[ \sum_{i : I_i = 1} \max c_i \leq e_0 \right] \land \neg L_i \Rightarrow \left[ \sum_{i : I_i = 1} e_i \leq x_i \right] \land \neg L_i \Rightarrow P \]

is in fact a smart instantiation of the clause

\[ \forall \tau: \left( \left[ \sum_{i} x_i \leq e_0 \right] \land \bigwedge_{i : I_i = 1} [e_i \leq x_i] \land \bigwedge_{i : I_i = 1} \neg L_i \right) \Rightarrow P \]

This sentence expresses that \( P \) is true if the sum of the maxima of all terms in \( S \), of which the condition is not false, is lower or equal than \( e_0 \). The other sentences are similar in idea, but not presented here. Similarly to handling comparison constraints, propagation is checked for the bounds of all terms and for all assignments to the associated atoms.

### F. General ground atoms

Constraints of the form \( P \iff q(\bar{x}) \) and \( P \iff f(\bar{x}) \sim e_0 \) are handled by waiting until all domain terms in \( \bar{x} \) are assigned. At that moment, the instantiated constraint is generated, which coincides with instantiations of the sentence

\[ \forall \tau \in \text{dom} e: [\tau = \tau] \Rightarrow (P \iff Q(\tau)), \text{respectively} \]

The propagation rule \textbf{Encode} \(_{\text{general}} \) adds the above sentences whenever the value of each of the \( c_i \) is known (applying Tseitin introduction to generate sentences in GNF). The former results in a set of clauses, the latter in a clause and a comparison constraints, which are both constraints of a lower type than the original sentence.

Constraints of the form \( P \iff f(\bar{x}) \sim e_0 \) are in fact a generalisation of the \textit{element} constraint from the field of Constraint Programming\([17]\), as the next example shows.

#### Example IV.3

An element constraint \textit{element}(c, A, i) expresses that a cp-variable (or constant) \( c \) takes the value at the index \( i \) of array \( A \). It is well-known that an array is in fact a function \( f_A \) from indices to values. The element constraint can then be modelled as the sentence \( f_A(i) = c \) and handled lazily as described above, by generating the comparison constraint \( f_A(d) = c \) when \( i \) is assigned to \( d \) in \( I \). It is possible that \( A \) (\( f_A \)) is very large or not completely known in advance.

Obviously, it is sometimes possible to derive propagation even before \( \tau \) is completely instantiated. Investigating the benefit of such propagation is part of future work.

### G. Definitions with function terms

In the standard case (no function terms), definitions are handled by applying the rules \textbf{Completion, Unfounded} and \textbf{Wellfounded}. Definitions containing function terms should be handled carefully, for which we introduce the extended rules \textit{Completion'}, \textit{Unfounded'} and \textbf{Wellfounded'}.
Consider a definition \( \Delta \) defining, among others, the symbol \( P \) by the rules \( \{ P(e_1) \leftarrow \varphi_1, \ldots, P(e_n) \leftarrow \varphi_n \} \). The completion of \( P \) for \( \Delta \) is the (non-ground) sentence
\[
\forall \tau : P(\tau) \Leftrightarrow \left( \bigvee_{i \in [1,n]} [\tau = e_i] \land \varphi_i \right)
\]
The rule \textit{Completion} adds the equivalent sentences
\[
\bigwedge_{i \in [1,n]} \varphi_i \Rightarrow P(e_i) \quad \forall \tau : P(\tau) \Rightarrow \left( \bigvee_{i \in [1,n]} [\tau = e_i] \land \varphi_i \right)
\]
The former sentence is added eagerly for each \( i \) (as it is already ground). For the latter sentence, \textit{Completion} adds its instantiation of \( \tau \) with \( d \) to \( T_s \) for atoms \( P(d) \) true in \( I \).

An issue with the condition on instantiation is that propagations might be missed. Indeed, the latter sentence of \textit{Completion} is only instantiated for \( P(d) \) true in \( I \); however, if \( \left( \bigvee_{i \in [1,n]} [\tau = e_i] \land \varphi_i \right) \) is false, then \( \neg P(d) \) is entailed. If \( P(d) \) does not occur in \( T_s \) (and is never added by other rules), it will not be decided, resulting in an interpretation of which not all two-valued extensions are models. It is easy to show that in a (non-failed) state in which no more inference rules are applicable, all unassigned domain atoms over defined symbols have to be false. Extending the interpretation in this way, denoted as the rule \textit{Defined-false}, then restores the post-condition of the algorithm.

For \textit{Unfounded} and \textit{Wellfounded}, we take an approach similar to previous sections: both rules are only applied when all domain terms occurring in \( \Delta \) are assigned. In such situations, replacing all domain terms in \( \Delta \) with their interpretation results in a definition to which the existing propagation rules \textit{Unfounded} and \textit{Wellfounded} can be applied. If one of these generates an explanation clause \( C \), this clause is only valid conditionally, as we had to substitute several constants in order to obtain it. So instead of adding \( C \) to \( T_s \), we add
\[
\bigwedge_{c \in \mathbb{C} \text{ occurs in } \Delta} [c = c^2] \Rightarrow C.
\]

**Example IV.4.** Consider part of a graph application consisting of a function \textit{next} mapping nodes to nodes and a constant start of type node. Suppose the aim is to compute a relation \( r \) on nodes, representing all nodes reachable from the start node through \textit{next}. The following is a definition of \( r \):
\[
\{ \begin{align*}
    r(\text{start}), & \quad \forall x : r(\text{next}(x)) \leftarrow r(x). \\
\end{align*}
\]
In the context of an interpretation \( I \) over domain \( \{a, b, c\} \), with \( \text{start}^2 = a \) and \( \text{next}^2 = \{a \mapsto b, b \mapsto a, c \mapsto c\} \), the definition reduces to the following definition, to which \textit{Unfounded} can be applied:
\[
\{ \begin{align*}
    r(a), & \quad r(b) \leftarrow r(a). \\
    r(a) & \leftarrow r(b). \\
    r(c) & \leftarrow r(c). \\
\end{align*}
\]
\textit{Unfounded} would then derive the unit clause \( \neg r(c) \); consequently, \textit{Unfounded} generates the clause
\[
(\text{start} = a) \land [\text{next}(a) = b] \land [\text{next}(b) = a] \land [\text{next}(c) = c]) \Rightarrow \neg r(c).
\]

\[\text{Note that the size of the grounding of this definition is linear in the size of the domain, instead of quadratic if functions would be graphed.}\]

**H. Pre-interpretation over some symbols**

As discussed in the section on grounding, next to the ground theory, the algorithm maintains a partial, symbolic interpretation \( I_{in} \), which is guaranteed to be consistent. However, we do not want to add this interpretation as constraints to the theory, for the same reason as we do not want to eagerly generate the full propositional grounding; e.g., if very few atoms over an interpreted predicate occur in \( T_s \). Instead, the following propagation rule takes care of adding just enough of \( I_{in} \) to obtain interpretations that are consistent with \( I_{in} \). Rule \textit{check-}\( I_{in} \) adds a clause \( \neg A \) to \( T_s \) for every atom \( A \) in \( T_s \) such that \( A^2_{in} \neq 0 \).

**Example IV.5.** Consider a theory \( T_s \) with constraint \( P(c) \land \neg P(c') \), with \( P \) over a large domain \( D \) and interpreted in \( I_{in} \). Adding clauses \( \neg P(d) \) for every domain-element \( d \in D \) would cause an immense grounding. However, lazily adding this whenever a value for \( c \) or \( c' \) is chosen, results in a theory where only the relevant literals are asserted.

**I. Complete search algorithm**

Next to the set of inference rules, a search algorithm consists of an execution order \( \ll \) on its rules. For the \textsc{Minisat(ID)} algorithm, the order is \texttt{Fail} \( \ll \) \texttt{Learn} \( \ll \) \texttt{UP} \( \ll \) \texttt{Unfounded} \( \ll \) \texttt{Decide} \( \ll \) \texttt{Wellfounded}. This order was chosen with efficiency in mind. E.g., whenever \texttt{Fail} is possible, it is useless to propagate further; \texttt{UP} is preferred over \texttt{Unfounded} because it is much cheaper and often derives more propagation; etc.

An additional concern is to not generate the same expression lazily multiple times, preferably without having to explicitly keep track of this. The approach taken is to order the rules in the inverse order of the type of constraints they apply to. This results in the following priority order (recall that \texttt{Encode}_{\text{max}} \) is executed in the initial phase).
\[
\texttt{Fail} \ll \texttt{Learn} \ll \texttt{UP} \ll \texttt{check-}I_{in} \ll \texttt{Encode} \ll \texttt{Comparison} \ll \textit{Completion} \ll \texttt{Encode}_{\text{sum}} \ll \texttt{Encode}_{\text{general}} \ll \textit{Unfounded} \ll \texttt{Decide} \ll \textit{Wellfounded} \ll \textit{Defined-false}
\]

**Theorem IV.6 (Soundness and completeness).** For any GNF theory \( T \) and consistent interpretation \( I_{in} \) over \( \Sigma(\bar{T}) \), the algorithm terminates and returns an interpretation \( I \), consistent with \( I_{in} \), such that all two-valued extensions of \( I \cup I_{in} \) are models of \( T \), or \texttt{fail} if no models of \( T \) exist that extend \( I_{in} \).

The proof is omitted due to lack of space.

**V. Experiments**

The grounding algorithm is implemented in the \texttt{IDP}\(^\dagger\) system\[^4\], a knowledge-base system supporting state-of-the-art model expansion, as can be observed from previous ASP competitions \[^{10},[5]\]. The search algorithm is implemented in the solver \textsc{Constraint-(ID)}, extending the state-of-the-art algorithm \textsc{Minisat(ID)} \[^7\]. As benchmarks, we used the benchmarks of the fourth ASP competition\(^8\) in the NP

\[^4\] By definition, \texttt{check-}I_{in} \) checks well-typedness of expressions.
\[^5\] Results of the fourth ASP competition are not available as of this writing.
\[^6\] Available at https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite
Experimental results show that the described techniques are a significant improvement, with almost no instances negatively affected.

VI. RELATED WORK AND CONCLUSION

The presented work fits in a more general effort to combine techniques from SAT, CP and high-level knowledge representation languages. The solver-independent CP language Zinc [23] is grounded to the language MiniZinc [25], supported by a range of search algorithms using various paradigms, as can be seen on www.minizinc.org/challenge2012/results2012.html. In the context of CSP, several systems ground to ASP extended with constraint atoms, such as Clingo [28] and EZ(CSP) [2]. For search, Clingo combines the ASP solver Clasp [13] with the CSP solver Gecode [15], while EZ(CSP) combines an off-the-shelf ASP solver with an off-the-shelf CLP-Prolog system. The prototype CSP solver Inca [12] searches for answer sets of a ground CASP program by applying lazy clause generation for arithmetic and all-different constraints. As opposed to extending the search algorithm, a different approach is to transform a CASP program to a pure ASP program [11], afterwards applying any off-the-shelf ASP solver. CASP languages generally only allow a restricted set of expressions to occur in constraint atoms and impose conditions on where constraint atoms can occur. For example, none of the languages allows general atoms $P(\tau)$ with $P$ an uninterpreted predicate symbol. One exception is the language $AC(\mathcal{C})$ by [24], a language aimed at integrating ASP and Constraint Logic Programming. As shown in [19], the language captures the languages of both Clingo and EZ(CSP); however, existing implementations only implement subsets of the language [16].

The presented ideas only improve performance when function symbols are present in the input theory. However, modellers are free to use predicates when some of its arguments depend functionally on each other and might choose to do so for example out of preference or ignorance of the functional relationship. In [6], it is investigated how functional relationships can be detected automatically, using a technique based on theorem proving, and how to subsequently rewrite the theory to introduce function symbols. Interesting avenues of future work are an experimental comparison with the above-mentioned systems and to investigate the effect of improving propagation for rules such as $Encode_{general}$ and $Unfolded$, which now only fire when most terms are assigned.

In this paper, we first presented a FO(·) grounding algorithm, parametrized by the function symbols allowed in the grounding. In this way, we can, without changes to the input language, support the next generation of search algorithms that integrate techniques from SAT, ASP and CP. Second, we presented a search algorithm for the ground fragment of FO(·). To the best of our knowledge, this is the first implementation for the full ground fragment of FO(·) (combining definitions with nested uninterpreted functions), which is one of the first freely available implementations of lazy clause generation. Experimental results show that the grounding size can be significantly reduced while obtaining similar or improved search performance.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># inst.</th>
<th># solved</th>
<th>avg. time/sec</th>
<th>avg. size (# atoms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bottle fill.</td>
<td>12</td>
<td>12(12)</td>
<td>39.5(39.4)</td>
<td>$6 \cdot 10^5(7 \cdot 10^5)$</td>
</tr>
<tr>
<td>crossing min.</td>
<td>7</td>
<td>7(7)</td>
<td>372.3(340.5)</td>
<td>$6 \cdot 10^5(3 \cdot 10^5)$</td>
</tr>
<tr>
<td>graceful graphs</td>
<td>11(3)</td>
<td>12(12)</td>
<td>387.1(244.5)</td>
<td>$1 \cdot 10^5(3 \cdot 10^5)$</td>
</tr>
<tr>
<td>incr. sched.</td>
<td>20(6)</td>
<td>20(20)</td>
<td>0.8(---)</td>
<td>$1 \cdot 10^4(---)$</td>
</tr>
<tr>
<td>no-mystery</td>
<td>8(8)</td>
<td>8(8)</td>
<td>2.9(0.8)</td>
<td>$1 \cdot 10^5(1 \cdot 10^5)$</td>
</tr>
<tr>
<td>pattern matching</td>
<td>6</td>
<td>6(6)</td>
<td>0.4(58.2)</td>
<td>$2 \cdot 10^4(2 \cdot 10^5)$</td>
</tr>
<tr>
<td>ricotter robots</td>
<td>5(5)</td>
<td>5(5)</td>
<td>267.7(269.4)</td>
<td>$9 \cdot 10^5(9 \cdot 10^5)$</td>
</tr>
<tr>
<td>sokoban</td>
<td>7(7)</td>
<td>7(7)</td>
<td>9.9(4.5)</td>
<td>$5 \cdot 10^5(6 \cdot 10^5)$</td>
</tr>
<tr>
<td>solitaire</td>
<td>20(14)</td>
<td>20(20)</td>
<td>26.1(32.2)</td>
<td>$3 \cdot 10^5(3 \cdot 10^5)$</td>
</tr>
<tr>
<td>stable marriage</td>
<td>20(20)</td>
<td>20(20)</td>
<td>68.5(66.3)</td>
<td>$2 \cdot 10^5(2 \cdot 10^5)$</td>
</tr>
<tr>
<td>still life</td>
<td>7(4)</td>
<td>4(4)</td>
<td>19.9(20.2)</td>
<td>$2 \cdot 10^5(6 \cdot 10^5)$</td>
</tr>
<tr>
<td>valve location</td>
<td>5(5)</td>
<td>5(5)</td>
<td>5.8(10.8)</td>
<td>$1 \cdot 10^6(2 \cdot 10^5)$</td>
</tr>
<tr>
<td>disjoint scheduling</td>
<td>20</td>
<td>20(20)</td>
<td>2.0(27.5)</td>
<td>$3 \cdot 10^5(2 \cdot 10^5)$</td>
</tr>
<tr>
<td>packing</td>
<td>20(4)</td>
<td>20(4)</td>
<td>0.7(279.0)</td>
<td>$1 \cdot 10^6(6 \cdot 10^5)$</td>
</tr>
</tbody>
</table>

TABLE 1. EXPERIMENTAL EVALUATION

(10)Experiments were run on a 64-bit Ubuntu 12.04 system with an Intel Core i5 3570 processor and 8 GB of RAM. All experimental data is available at dta.cs.kuleuven.be/kmr/research/experiments
REFERENCES