

**Proceedings of the CP 2004 Workshop on
CSP Techniques with Immediate Application
(CSPIA)**

Roman Barták
Ulrich Junker
Marius-Calin Silaghi
Markus Zanker
(editors)

**Held at the
International Conference on Principles and Practice of
Constraint Programming
CP-2004
September 27th, 2004
Toronto, Canada**

ORGANISATION

Organizers and Workshop Chairs

Roman Barták, Charles University,
bartak@kti.mff.cuni.cz

Ulrich Junker, ILOG,
junker@ilog.fr

Marius-Calin Silaghi, Florida Institute of Technology,
marius.silaghi@cs.fit.edu

Markus Zanker, University Klagenfurt,
markus.zanker@ifit.uni-klu.ac.at

Program Committee

Roman Barták, Charles University,
bartak@kti.mff.cuni.cz

Frederic Benhamou, University of Nantes,
Frederic.Benhamou@irin.univ-nantes.fr

Berthe Choueiry, University Nebraska Lincoln,
choueiry@cse.unl.edu

Joerg Denzinger, University Calgary,
denzinge@cpsc.ucalgary.ca

Boi Faltings, EPFL,
Boi.Faltings@epfl.ch

Alexander Felfernig, University Klagenfurt,
alexander.felfernig@ifit.uni-klu.ac.at

Christian Frei, ABB Switzerland,
christian.frei@ch.abb.com

Gerhard Friedrich, University Klagenfurt,
gerhard@ifit.uni-klu.ac.at

Dietmar Jannach, University Klagenfurt,
dietmar@ifit.uni-klu.ac.at

Pedro Meseguer, Institut d'Investigació en Intelligència Artificial (IIIA),
pedro@iiia.csic.es

Debasis Mitra, Florida Institute of Technology,
dmitra@cs.fit.edu

Christian Russ, ConfigWorks Informationssysteme & Consulting, Austria,
cruss@configworks.com

Markus Stumptner, University of South Australia,
markus.stumptner@unisa.edu.au

Barry O'Sullivan, University College Cork,
b.osullivan@cs.ucc.ie

Marc Wallace, Monash University,
Mark.Wallace@infotech.monash.edu.au

Franz Wotawa, Technical University Graz,
fwotawa@ist.tu-graz.ac.at

PREFACE

Constraint satisfaction is a successful paradigm developed by the AI community that found acceptance in many fields of computer science. While specific techniques (e.g. operations research ones) exist for most of the problems that can be represented as Constraint Satisfaction Problems (CSPs), constraint satisfaction owes its success to the simplicity and straightforwardness with which humans can formalize new problems and profit thereby of powerful existing techniques.

While there exist extensive literature about abstract techniques for solving more or less general CSP formulations, the applicability of the field is covered by a veil of mystery. The companies that propose CSP based solutions seem to have particular success on the market but the details and the contribution of the Constraint Satisfaction research to all of it are all but well known. The lack of feedback to the large amount of academic researchers involved in the field encourages a more in breadth research versus a focused research.

Moreover, most conferences in the field give currently only little attention to the relation technique/application and focus solely on randomly generated or on toy evaluation criteria. Most problems faced by the researchers trying to apply constraint satisfaction to real applications are invisible to the community as they do not find easily an avenue into the main conferences like CP. The series of applied science - workshops was started at CP 2003 and intends to foster a better interaction between physical reality and academic research on CSPs and to direct CSP research towards areas of high promise and social interest. (All the difference is BFS vs. A*!)

This goal of interaction between industry and academia resembles also in the choice of the *invited speaker*:

Mark Wallace addresses the issue when industry meets research and discusses three successful application areas in transport industry.

The working notes of this workshop gather contributions from applications such as interactive configuration, computer graphics, order sequencing and model checking of UML diagrams. Furthermore, research on search techniques and DisCSPs with a focus on its applicability is presented. The 8 papers demonstrate that continuous technical advances in CSPs lead to a technology that fits more and more real applications.

Roman Barták
Ulrich Junker
Marius-Calin Silaghi
Markus Zanker

Aug. 25, 2003

CONTENTS

<i>Invited talk: Three Research Collaborations with the Transport Industry</i> Mark Wallace	1
Finite Satisfiability of UML class diagrams by constraint programming Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, Toni Mancini	2-16
Interval Constraints for Computer Graphics Belaid Moa	17-36
Multiplex Dispensation Order Generation for Pyrosequencing Mats Carlsson and Nicolas Beldiceanu	37-51
Boosting Constraint Satisfaction using Decision Trees Barry O'Sullivan, Alex Ferguson, Eugene C. Freuder	52-65
An empirical Study of Heuristic and Randomized Search Techniques in a Real-World Setting Venkata Praveen Guddeti, Hui Zou, Berthe Y. Choueiry	66-82
Desk-mates (Stable Matching) with Privacy of Preferences and a new Distributed CSP Framework Marius-Calin Silaghi, Markus Zanker, Roman Barták	83-96
Comparing two implementations of a Complete and Backtrack-free Interactive Configurator Sathiamoorthy Subbarayan, Rune M. Jensen, Tarik Hadzic, Henrik R. Andersen, Henrik Hulgaard, Jesper Moller	97-111
BBD-based Recursive and Conditional Modular Interactive Product Configuration Erik R. van der Meer, Henrik Reif Andersen	112-126

Three Research Collaborations with the Transport Industry

Mark Wallace

Faculty of Information Technology
Monash University
Building 63, Clayton
Vic 3800
Australia
`Mark.Wallace@infotech.monash.edu.au`

Abstract. The talk will explore the three problem definitions, and give their business motivation. The projects were about

- logistics with depots,
- patrol dispatcher and
- flight schedule retimer.

The talk will then discuss the algorithms used to solve them, and hopefully also give an insight into the benefits of applications-driven research.

Finite Satisfiability of UML class diagrams by Constraint Programming

Marco Cadoli¹, Diego Calvanese², Giuseppe De Giacomo¹, Toni Mancini¹

¹ Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, I-00198 Roma, Italy
`cadoli|degiacomo|tmancini@dis.uniroma1.it`
`www.dis.uniroma1.it/~cadoli|~degiacomo|~tmancini`
² Faculty of Computer Science
Free University of Bolzano/Bozen
Piazza Domenicani 3, I-39100 Bolzano, Italy
`calvanese@inf.unibz.it`

Abstract. Finite model reasoning in UML class diagrams, e.g., checking whether a class is forced to have either zero or infinitely many objects, is of crucial importance for assessing quality of the analysis phase in software development. Despite the fact that finite model reasoning is often considered more important than unrestricted reasoning, no implementation of the former task has been attempted so far. The main result of this paper is that it is possible to use off-the-shelf tools for constraint modeling and programming for obtaining a finite model reasoner. In particular, exploiting appropriate reasoning techniques, we propose an encoding as a CSP of UML class diagram satisfiability. Moreover, we show also how CP can be used to actually return a finite model of a class diagram. A description of our system, which accepts as input class diagrams in the MOF syntax, and the results of the experimentation performed on the CIM knowledge base are given.

1 Introduction

The Unified Modelling Language (UML, [13], cf. www.uml.org) is probably the most used modelling language in the context of software development, and has been proven to be very effective for the analysis and design phases of the software life cycle.

UML offers a number of diagrams for representing various aspects of the requirements for a software application. Probably the most important diagram is the *class diagram*, which represents all main structural aspects of an application. A typical class diagram shows:

- *classes*, i.e., homogeneous collections of *objects*, i.e., instances;
- *associations*, i.e., relations between classes;
- *ISA hierarchies* between classes, i.e., relations establishing that each object of a class is also an object of another class;

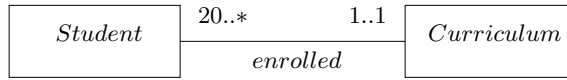


Fig. 1. A UML class diagram.

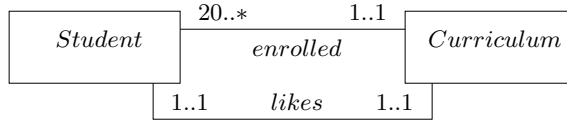


Fig. 2. A UML class diagram with finitely inconsistent classes.

- *multiplicity constraints* on associations, i.e., restrictions on the number of links between objects related by an association.

Actually, a UML class diagram represents also other aspects, e.g., the attributes and the operations of a class, the attributes of an association, and the specialization of an association. Such aspects, for the sake of simplicity, will not be considered in this paper.

An example of a class diagram is shown in Figure 1, which refers to an application concerning management of administrative data of a university, and exhibits two classes (*Student* and *Curriculum*) and an association (*enrolled*) between them. The multiplicity constraints state that:

- Each student must be enrolled in at least one and at most one curriculum;
- Each curriculum must have at least twenty enrolled students, and there is no maximum on the number of enrolled students per curriculum.

It is interesting to note that a class diagram induces restrictions on the number of objects. As an example, referring to the situation of Figure 1, it is possible to have zero, twenty, or more students, but it is impossible to have any number of students between one and nineteen. The reason is that if we had, e.g., five students, then we would need at least one curriculum, which in turn requires at least twenty students.

In some cases the number of objects of a class is forced to be zero. As an example, if we add to the class diagram of Figure 1 a further *likes* association, with the constraints that each student likes exactly one curriculum, and that each curriculum is liked by exactly one student (cf. Figure 2), then it is impossible to have any finite non-zero number of students and curricula. In fact, the new association and its multiplicity constraints force the students to be the exactly as many as the curricula, which is impossible. Observe that, with a logical formalization the UML class diagram, one can actually perform such form of reasoning making use of automated reasoning tools¹.

¹ Actually, current CASE tools do not perform any kind of automated reasoning on UML class diagrams yet.

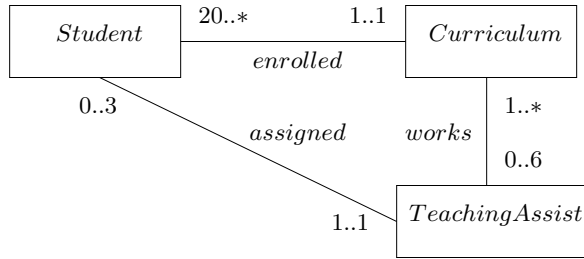


Fig. 3. Another finitely inconsistent class diagram.

Referring to Figure 2, note that the multiplicity constraints do not rule out the possibility of having *infinitely many* students and curricula. When a class is forced to have either zero or infinitely many instances, it is said to be *finitely inconsistent* or *finitely unsatisfiable*. For the sake of completeness, we mention that in some situations involving ISA hierarchies (not shown for brevity), classes may be forced to have zero objects, and are thus said to be inconsistent or unsatisfiable in the *unrestricted* sense.

Unsatisfiability, either finite or unrestricted, of a class is a symptom of a bug in the analysis phase, since such a class is clearly superfluous. In particular, finite unsatisfiability is especially relevant in the context of applications, e.g., databases, in which the number of instances is intrinsically finite.

Obviously the situation described by Figure 2 (in particular, the fact that a curriculum is liked by exactly one student) is not realistic. Anyway, finite inconsistency may arise in more complex situations, cf. e.g. Figure 3. Here, for budget reasons, each curriculum has at most six teaching assistants, and, for enforcing effectiveness of service, each TA is assigned to at most three students. The reason why there must be zero (or infinitely many) students is that eighteen, i.e., three times six, is less than twenty. Global reasoning on the whole class diagram is needed to show finite inconsistency. For large, industrial class diagrams, such reasoning is clearly not doable by hand.

If, in our application we have that the UML class diagram will always be instantiated by a finite number of objects (as often is the case), then, for assessing quality of the analysis phase in software development, we must take fully into account such an assumption. This is reflected in focusing on finite model reasoning.

In this paper we address the implementation of finite model reasoning on UML class diagrams, a task that has not been attempted so far. This is done by exploiting an encoding of UML class diagrams in terms of Description Logics (DLs) [6, 4].

DLs [1] are logics for representing and reasoning on domains of interest in terms of classes and relationships among classes. They are extensively used to formalize conceptual models and object-oriented models in databases and software engineering, and lay the foundations for ontology languages used in the Semantic Web. From a technical point of view, DLs can be seen as multi-modal

logics [16] specifically tailored to capture the typical constructs of class-based formalisms. Alternatively, they can be seen as well-behaved fragments of first-order logic. Indeed, reasoning in such logics has been studied extensively, both from a theoretical point of view, establishing EXPTIME-completeness of various DL variants [9], and from a practical point of view, developing practical reasoning systems. State-of-the-art DL reasoning systems, such as FACT² and RACER³, are highly optimized and result among the best reasoners for modal logics.

The correspondence between UML class diagrams and DLs allows one to use the current state-of-the-art DL reasoning systems to reason on UML class diagrams. However, the kind of reasoning that such systems support is unrestricted (as in first-order logic), and not finite model reasoning. That is, the fact that models (i.e., instantiations of the UML class diagram) must be finite is not taken into account.

Interestingly, in DLs, finite model reasoning has been studied from a theoretical perspective, and its computational complexity has been characterized for various cases [14, 10, 7, 15]. However, no implementations of such techniques have been attempted till now. In this paper we reconsider such work, and on the basis of it we present, to the best of our knowledge, the first implementation of finite model reasoning in UML class diagrams.

The main result of this paper is that it is possible to use off-the-shelf tools for constraint modelling and programming for obtaining a finite model reasoner. In particular, exploiting the finite model reasoning technique for DLs presented in [10, 7], we propose an encoding of UML class diagram satisfiability as a Constraint Satisfaction Problem (CSP). Moreover, we show also how constraint programming can be used to actually return a finite model of the UML class diagram.

We built a system that accepts as input a class diagram written in the MOF syntax, and translates it into a file suitable for ILOG's OPLSTUDIO, which checks satisfiability and returns a finite model, if there is one. The system allowed us to test the technique on the industrial knowledge base CIM, obtaining encouraging results.

The rest of the paper is organized as follows: in Section 2 we briefly describe the main constructs of UML class diagrams in terms of first-order logic. In Sections 3 and 4 we then show how to encode the UML class diagram finite satisfiability problem as a CSP, and, in Section 5 we show how to find, if possible, a finite model of a class diagram, with non-empty classes and associations. Section 6 is devoted to some observations on complexity issues, while our system is described in Section 7. Finally, Section 8 concludes the paper.

2 Formalization of UML Class Diagrams

UML class diagrams allow for modelling, in a declarative way, the static structure of an application domain, in terms of concepts and relations between them.

² <http://www.cs.man.ac.uk/~horrocks/FaCT/>

³ <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>

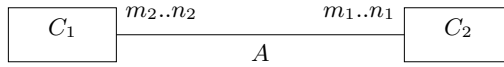


Fig. 4. Binary association in UML with multiplicity constraints.

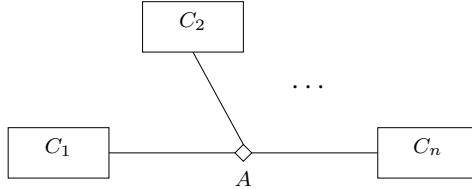


Fig. 5. n -ary association in UML.

We briefly describe UML class diagrams, and specify the semantics of the main constructs in terms of first-order logic (FOL).

A *class* in a UML class diagram denotes a set of objects with common features. Names of classes are unique in a UML class diagram. Formally, a class C corresponds to a FOL unary predicate C . Classes may have attributes and operations, but for simplicity we do not describe them here, since they have only minor impact on the reasoning process.

An *association* in UML is a relation between the instances of two or more classes. Names of associations are unique in a UML class diagram. A binary association A between two classes C_1 and C_2 is graphically rendered as in Figure 4. The *multiplicity* $m_1..n_1$ on the binary association specifies that each instance of the class C_1 can participate at least m_1 times and at most n_1 times to A , similarly for C_2 . When the multiplicity is omitted, it is intended to be $0..*$. Observe that an association can also relate several classes C_1, C_2, \dots, C_n , as depicted in Figure 5⁴.

An association A between the instances of classes C_1, \dots, C_n , can be formalized as an n -ary predicate A that satisfies the following FOL assertion:

$$\forall x_1, \dots, x_n. A(x_1, \dots, x_n) \rightarrow C_1(x_1) \wedge \dots \wedge C_n(x_n)$$

For binary associations (see Figure 4), multiplicities are formalized by the FOL assertions:

$$\begin{aligned} \forall x. C_1(x) &\rightarrow (m_1 \leq \#\{y \mid A(x, y)\} \leq n_1) \\ \forall y. C_2(y) &\rightarrow (m_2 \leq \#\{x \mid A(x, y)\} \leq n_2) \end{aligned}$$

where we have abbreviated FOL formulas expressing cardinality restrictions.

Aggregations, which are a particular kind of binary associations are modeled similarly.

⁴ In UML, differently from other conceptual modelling formalisms, such as Entity-Relationship diagrams [2], multiplicities are look-across cardinality constraints [17]. This makes their use in non-binary associations difficult with respect to both modelling and reasoning.

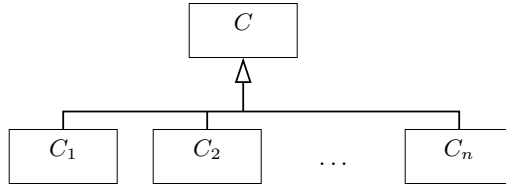


Fig. 6. A class hierarchy in UML.

In UML one can use a *generalization* between a parent class and a child class to specify that each instance of the child class is also an instance of the parent class. Hence, the instances of the child class inherit the properties of the parent class, but typically they satisfy additional properties that in general do not hold for the parent class. Several generalizations can be grouped together to form a *class hierarchy* (also called *ISA hierarchy*), as shown in Figure 6. *Disjointness* and *completeness constraints* can also be enforced on a class hierarchy (graphically, by adding suitable labels). A class hierarchy is said to be disjoint if no instance can belong to more than one derived class, and complete if any instance of the base class belongs also to some of the derived classes.

A UML class C generalizing a class C_1 can be formally captured by means of the FOL assertion:

$$\forall x. C_1(x) \rightarrow C(x)$$

A class hierarchy as the one in Figure 6 is formally captured by means of the FOL assertions:

$$\forall x. C_i(x) \rightarrow C(x), \quad \text{for } i = 1, \dots, n$$

Disjointness among C_1, \dots, C_n is expressed by the FOL assertions

$$\forall x. C_i(x) \rightarrow \bigwedge_{j=i+1}^n \neg C_j(x), \quad \text{for } i = 1, \dots, n-1$$

The *completeness constraint* expressing that each instance of C is an instance of at least one of C_1, \dots, C_n is expressed by:

$$\forall x. C(x) \rightarrow \bigvee_{i=1}^n C_i(x)$$

In UML class diagrams, it is typically assumed that all classes not in the same hierarchy are a priori disjoint. Similarly, it is typically assumed that objects in a hierarchy must belong to a single most specific class. Hence, two classes in a hierarchy may have common instances only if they have a common subclass.

3 Finite Model Reasoning on UML Class Diagrams

As mentioned before, a technique for finite model reasoning in UML class diagrams can be derived from techniques developed in the context of Description Logics (DLs) [1]. Such techniques are based on translating a DL knowledge base

into a set of linear inequalities [10, 7]. The first-order formalization of UML class diagrams shown in the previous section can be rephrased in terms of DLs. Hence, the finite model reasoning techniques for DLs can be used also for UML class diagrams.

Intuitively, consider a simple UML class diagram D with only binary associations, and in which we do not make use of generalization and hierarchies. Further, for each association, between two classes multiplicities are specified. Figure 4 shows a fragment of such a diagram, in which we have two classes C_1 and C_2 and an association A between them. The multiplicities in the figure express that each instance of C_1 is associated with at least m_1 and at most n_1 instances of C_2 through the association A , similarly for C_2 . It is easy to see that such a class diagram D is always satisfiable (assuming $m_i \leq n_i$) if we admit infinite models. Hence, only finite model reasoning is of interest. We observe that, if D is finitely satisfiable, then it admits a finite model in which all classes are pairwise disjoint. Exploiting this property, we can encode finite satisfiability of D in a constraint system as follows. We introduce one variable for each class and one for each association, representing the number of instances of the class (resp., association) in a possible model of D . Then, for each association A we introduce the constraints:

$$\begin{aligned} m_1 * c_1 &\leq a \leq n_1 * c_1 \\ m_2 * c_2 &\leq a \leq n_2 * c_2 \\ c_1 * c_2 &\geq a \end{aligned}$$

where c_1 , c_2 , and a are the variables corresponding to C_1 , C_2 , and A , respectively.

It is possible to show that, from a solution of such a constraint system we can construct a finite model of D in which the cardinality of the extension of each class and association is equal to the value assigned to the corresponding variable⁵ [14].

The above approach can be extended to deal also with generalizations, disjointness, and covering between classes. Intuitively, one needs to introduce one variable for each combination of classes; similarly, for associations one needs to distinguish how, among the possible combinations of classes, the association is typed in its first and second component. This leads, in general, to the introduction of an exponential number of variables and constraints [10, 7]. We illustrate this in the next section.

4 Finite Model Reasoning via CSP

We address the problem of finite satisfiability of UML class diagrams, and show how it is possible to encode two problems as constraint satisfaction problems (CSPs), namely:

1. deciding whether all classes in the diagram are simultaneously finitely satisfiable, and

⁵ In fact, if one is interested just in the existence of a finite model, one could drop the nonlinear constraints of the form $c_1 * c_2 \geq a$.

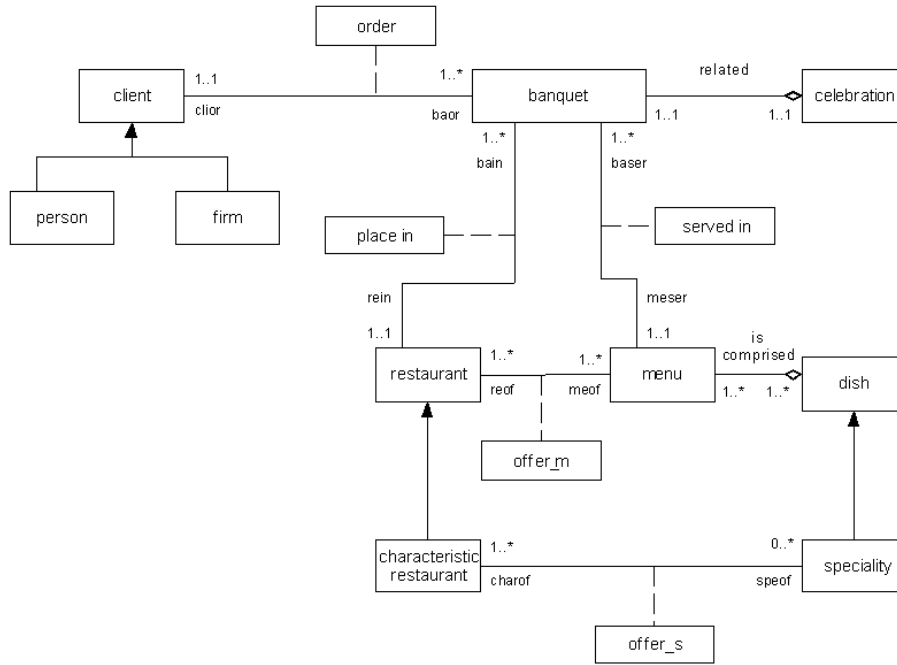


Fig. 7. The “restaurant” UML class diagram.

2. finding –if possible– a finite model with non-empty classes and associations.

We use the “restaurant” class diagram, shown in Figure 7, as our running example.

First we address the problem of deciding finite satisfiability. As mentioned before, we use the technique proposed in [7], which is based on the idea of translating the multiplicity constraints of the UML class diagram into a set of inequalities among integer variables.

The variables and the inequalities of the CSP are modularly described considering in turn each association of the class diagram. Let a be an association between classes $c1$ and $c2$ such that the following multiplicity constraints are stated:

- There are at least min_{c1} and at most max_{c1} links of type a (instances of the association a) for each object of the class $c1$;
- There are at least min_{c2} and at most max_{c2} links of type a for each object of the class $c2$.

Referring to Figure 7, if a stands for *served_in*, $c1$ stands for *banquet*, and $c2$ stands for *menu*, then min_{c1} is 1, max_{c1} is 1, min_{c2} is 1, and max_{c2} is ∞ .

For the sake of simplicity, we start from the special case in which neither $c1$ nor $c2$ participates in a ISA hierarchy, e.g., the *related* and the *served_in* associations of Figure 7.

The CSP is defined as follows:

- There are three non-negative variables c_1 , c_2 , and a , which stand for the number of objects of the classes and the number of links, respectively (in practice, upper bounds for these variables can be set to a huge constant, e.g., `maxint`);
- There are the following constraints (we use the syntax of the constraint programming language OPL [18]):
 1. `min_c1 * c1 <= a;`
 2. `max_c1 * c1 >= a;`
 3. `min_c2 * c2 <= a;`
 4. `max_c2 * c2 >= a;`
 5. `a <= c1 * c2;`
 6. `a >= 1;`
 7. `c1 >= 1;`
 8. `c2 >= 1;`

Constraints 1–4 account for the multiplicity of the association; they can be omitted if either $\text{min}_- = 0$, or $\text{max}_- = \infty$ (symbol ‘*’ in the class diagram). Constraint 5 sets an upper bound for the number of links of type a with respect to the number of objects. Constraints 6–8 define the satisfiability problem we are interested in: we want at least one object for each class and at least one link for each association. The latter constraints can be omitted by declaring the variables as strictly positive. Finally, to avoid the system returning an ineffectively large solution, an objective function that, e.g., minimizes the overall number of objects and links, may be added. However, the addition of such or other objective functions is out of the scope of this paper.

When either c_1 or c_2 are involved in ISA hierarchies, the constraints are more complicated, because the meaning of the multiplicity constraints changes. As an example, the multiplicity `1..*` of the *order* association in Figure 7 states that a *client* orders at least one *banquet*, but the client can be a *person*, a *firm*, both, or neither (assuming the generalization is neither disjoint nor complete). In general, for an ISA hierarchy involving n classes, $O(2^n)$ non-negative variables corresponding to all possible combinations must be considered. For the same reason, we must consider four distinct specializations of the *order* association, i.e., one for each possible combination. Summing up, we have the following non-negative variables:

- `person`, `order_p`, for clients who are persons and not firms;
- `firm`, `order_f`, for clients who are firms and not persons;
- `person_firm`, `order_pf`, for clients who are both firms and persons;
- `client`, `order_c`, for clients who are neither firms nor persons;

plus the positive `banquet` variable.

The constraints (in the OPL syntax) which account for the *order* association are as follows:

```

/* 1 */ client <= order_c;
/* 2 */ firm <= order_f;
/* 3 */ person <= order_p;
/* 4 */ person_firm <= order_pf;
/* 5 */ banquet = order_c + order_f + order_p + order_pf;
/* 6 */ order_c <= client * banquet;
/* 7 */ order_f <= firm * banquet;
/* 8 */ order_p <= person * banquet;
/* 9 */ order_pf <= person_firm * banquet;
/* 10 */ client + firm + person + person_firm >= 1;
/* 11 */ order_c + order_f + order_p + order_pf >= 1;

```

Constraints 1–4 account for the ‘1’ in the $1..*$ multiplicity; Constraint 5 translates the $1..1$ multiplicity; Constraints 6–9 set an upper bound for the number of links of type *order* with respect to the number of objects; Constraints 10–11 define the satisfiability problem (*banquet* is already strictly positive).

We refer the reader to [7, 8] for formal details of the translation, and in particular for the proof of its correctness. As for the implementation, the “restaurant” example has been encoded in OPL as a CSP with 24 variables and 40 constraints. The solution has been found by the underlying constraint programming solver, i.e., ILOG’s SOLVER [12, 11], in less than 0.01 seconds.

5 Constructing a Finite Model

We now turn to the second problem, i.e., finding –if possible– a finite model with non-empty classes and associations. The basic idea is to encode in the constraint modelling language of OPL the semantics of the UML class diagram (see Section 2 and [4, 3]). In particular we use arrays of boolean variables representing the extensions of predicates, where the size of the arrays is determined by the output of the first problem. Since in the first problem we have enforced the multiplicity constraints, and obtained an admissible number of objects for each class, we know that a finite model of the class diagram exists, and we also know the size of the universe of such a finite model, which is the sum of the objects of the classes.

Referring to our “restaurant” example, we have the following declarations describing the size of our universe and two sorts:

```

int size = client + person + firm + person_firm + restaurant + menu +
  characteristic_restaurant + dish + specialty + banquet + celebration;
range Bool 0..1;
range Universe 1..size;

```

where *client*, *person* etc. are the number of objects for each class, obtained as the output of the first problem.

The arrays corresponding, e.g., to the *client* and *banquet* classes, and to the *order_c* association are declared as follows:

```

var Bool Client[Universe];

```

```

var Bool Banquet[Universe];
var Bool Order_C[Universe,Universe];

```

Now, we have to enforce some constraints to reflect the semantics of the UML class diagram [4, 3], namely that:

1. Each object belongs to exactly one most specific class;
2. The number of objects (resp., links) in each class (resp., association) is coherent with the solution of the first problem;
3. The associations are *typed*, e.g., that a link of type *order_c* insists on an object which is a *banquet* and on another object which is a *client*;
4. The multiplicity constraints are satisfied.

Such constraints can be encoded as follows (for brevity, we show only some of the constraints).

```

// AN OBJECT BELONGS TO ONE CLASS
forall(x in Universe)
  Client[x] + Person[x] + Firm[x] + Person_Firm[x] + Restaurant[x] +
  Characteristic_Restaurant[x] + Dish[x] + Specialty[x] + Banquet[x] +
  Celebration[x] + Menu[x] = 1;
// ENFORCING SIZES OF CLASSES AND ASSOCIATIONS
sum(x in Universe) Client[x] = client;
sum(x in Universe) Banquet[x] = banquet;
sum(x in Universe, y in Universe) Order_C[x,y] = order_c;
// TYPES FOR ASSOCIATIONS
forall(x, y in Universe)
  Order_C[x,y] => Client[x] & Banquet[y];
// MULTIPLICITY CONSTRAINTS ARE SATISFIED
forall(x in Universe)
  Client[x] => sum(y in Universe) Order_C[x,y] >= 1;

```

Summing up, the “restaurant” example has been encoded in OPL with about 40 lines of code. After instantiation, this resulted in a CSP with 498 variables and 461 constraints. The solution has been found by ILOG’s SOLVER in less than 0.01 seconds, and no backtracking.

6 Notes on complexity

Few notes about the computational complexity are in order. It is known that solving both problems of deciding finite satisfiability and finite model finding are EXPTIME-complete [7]. Our encoding of the first problem in a CSP may result in a number of variables which is exponential in the size of the diagram. Anyway, since the exponentiality depends on the maximum number of classes involved in the same ISA hierarchy, the actual size for real UML diagrams will typically not be very large (especially when the most specific class assumption is enforced, cf. Section 2). As for the second problem, our encoding is polynomial in the size of the class diagram. Note that this does not contradict the EXPTIME

lower bound, due to the program complexity of modelling languages such as OPL. Indeed, in [5] it is shown that the program complexity of boolean linear programming is NEXPTIME-hard.

7 Implementation

In this section we describe a system realized in order to automatically produce, given a UML class diagram as input, a constraint-based specification that decides its finite satisfiability (full handling of ISA hierarchies among classes and associations is currently under development).

Two important choices have to be made in order to design such a system: the input language for UML class diagrams, and the output constraint language. As for the former, we decided to use a textual representation of UML class diagrams. To this end, we relied on the standard language “Managed Object Format” (MOF)⁶. To give the intuition of the language, a MOF description of the class diagram depicted in Figure 7 is shown in Figure 8. For what concerns the output language, instead, in order to use state-of-the-art solvers, we opted for the constraint programming language OPL.

However, in order to have a strong decoupling between the two front-ends of the system, we realized it in two, strongly decoupled modules: the first one acts as a server, receiving a MOF file as input and returning a high-level, object-oriented complete internal representation of the described class diagram. A client module, then, traverses the internal model in order to produce the OPL specification.

In this way, we are able to change the language for the input (resp., output) by modifying only the MOF parser (resp., the OPL encoder) module of the system. Moreover, by decoupling the parsing module from the encoder into OPL, we are able to realize new tools to make additional forms of reasoning at a very low cost.

As for the handling of ISA hierarchies, as described in Section 4, an exponential blow-up of the number of variables (one for each combination of classes involved in the hierarchy) cannot be avoided in general. However, in case the hierarchy is disjoint or complete, it is possible to reduce the number of generated variables. It can be observed that, if an ISA is complete, the variable relative to its base class can be avoided; even more interestingly, if an ISA is disjoint, all variables that model instances that belong to *any* combination of two or more derived classes can be ignored, thus reducing the overall number of variables to the number of classes in the hierarchy.

The MOF language provides the “abstract” qualifier that, when applied to a class C imposes that no instance may exist that belongs to class C and does not belong to any of its subclasses. This is implicitly used to assert completeness of the ISA hierarchy. Also, in MOF, the most specific class assumption is implicitly enforced. Hence, in all experiments reflect such an assumption.

In order to test whether using off-the-shelf tools for constraint programming is effective to decide finite satisfiability of real-world UML class diagrams, we

⁶ <http://www.dmtf.org>

used our system to produce OPL specifications for several class diagrams of the “Common Information Model” (CIM)⁷, a standard model used for describing overall management information in a network/enterprise environment. We don’t describe the model into details, since this is out of the scope of the paper. We just observe that the class diagrams we used were composed of about 1000 classes and associations, and so can be considered good benchmarks to test whether current constraint programming solvers can be effectively used to make the kind of reasoning shown so far.

Constraint specifications obtained by giving large class diagrams in the CIM collection, were solved very efficiently by OPL. As an example, when the largest diagram, consisting of 980 classes and associations, has been given as input to our system, we obtained an OPL specification consisting of a comparable number of variables and 862 constraints. Nonetheless, OPL solved it in less than 0.03 seconds of CPU time, by invoking ILOG SOLVER. This high efficiency is achieved also because generated constraints are often easily satisfiable (cardinality constraints for associations often have “0” or “1” as lower bounds, or “*” as upper bounds). This is encouraging evidence that current CP technology can be effectively used in order to make finite model reasoning on real-world class diagrams.

8 Conclusions

Finite model reasoning in UML class diagrams, e.g., checking whether a class is forced to have either zero or infinitely many objects, is of crucial importance for assessing quality of the analysis phase in software development. Despite the fact that finite model reasoning is often considered more important than unrestricted reasoning, no implementation of this task has been attempted so far.

In this paper we showed that it is possible to use off-the-shelf tools for constraint modelling and programming for obtaining a finite model reasoner. In particular, exploiting finite model reasoning techniques published previously, we proposed an encoding as a CSP of UML class diagram satisfiability. Moreover, we showed also how constraint programming can be used to actually return a finite model of a class diagram.

We implemented a system which parses class diagrams written in the MOF language and uses ILOG’s SOLVER for solving the finite satisfiability problem. The results of the experimentation performed on the CIM class diagram, a large industrial knowledge base, are encouraging, since determining satisfiability is done in just few hundredths of second for a class diagram with 980 classes and associations.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

⁷ <http://www.dmtf.org/standards/cim>

2. C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach*. Benjamin and Cummings Publ. Co., Menlo Park, California, 1992.
3. D. Berardi, A. Cali, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. Technical Report 11-03, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, 2003.
4. D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams is EXPTIME-hard. In *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, pages 28–37. CEUR Electronic Workshop Proceedings, <http://ceur-ws.org/Vol-81/>, 2003.
5. M. Cadoli. The expressive power of binary linear programming. In *Proc. of the 7th Int. Conf. on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, 2001.
6. A. Cali, D. Calvanese, G. De Giacomo, and M. Lenzerini. A formal framework for reasoning on UML class diagrams. In *Proceedings of the Thirteenth International Symposium on Methodologies for Intelligent Systems (ISMIS 2002)*, volume 2366 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2002.
7. D. Calvanese. Finite model reasoning in description logics. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 292–303, 1996.
8. D. Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms*. PhD thesis, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, 1996. Available at <http://www.dis.uniroma1.it/pub/calvanes/thesis.ps.gz>.
9. D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 23, pages 1581–1634. Elsevier Science Publishers (North-Holland), Amsterdam, 2001.
10. D. Calvanese and M. Lenzerini. On the interaction between ISA and cardinality constraints. In *Proceedings of the Tenth IEEE International Conference on Data Engineering (ICDE'94)*, pages 204–213, Houston (Texas, USA), 1994. IEEE Computer Society Press.
11. ILOG Solver system version 5.1 user’s manual, 2001.
12. ILOG OPL Studio system version 3.6.1 user’s manual, 2002.
13. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley Publ. Co., Reading, Massachusetts, 1998.
14. M. Lenzerini and P. Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems*, 15(4):453–461, 1990.
15. C. Lutz, U. Sattler, and L. Tendera. The complexity of finite model reasoning in description logics. In *Proceedings of the Nineteenth International Conference on Automated Deduction (CADE 2003)*, pages 60–74, 2003.
16. K. Schild. A correspondence theory for terminological logics: Preliminary report. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 466–471, 1991.
17. B. Thalheim. Fundamentals of cardinality constraints. In G. Pernoul and A. M. Tjoa, editors, *Proceedings of the Eleventh International Conference on the Entity-Relationship Approach (ER'92)*, pages 7–23. Springer, 1992.
18. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

```

class Client { /* Properties and methods (always omitted) */ };
class Person : Client {};
class Firm : Client {};
class Banquet {};
class Celebration {};
class Restaurant {};
class Menu {};
class Dish {};
class CharacteristicRestaurant : Restaurant {};
class Speciality : Dish {};

[association] class Order {
  [Min (1), Max (1)] Client REF clior;
  [Min (1)] Banquet REF baor;
};
[association] class Related {
  [Min (1), Max (1)] Celebration REF cerel;
  [Min (1), Max (1)] Banquet REF barel;
};
[association] class PlaceIn {
  [Min (1)] Banquet REF bain;
  [Min (1), Max (1)] Restaurant REF rein;
};
[association] class ServedIn {
  [Min (1)] Banquet REF baser;
  [Min (1), Max (1)] Menu REF meser;
};
[association] class IsComprised {
  [Min (1)] Menu REF mecom;
  [Min (1)] Dish REF dicom;
};
[association] class OfferM {
  [Min (1)] Restaurant REF reof;
  [Min (1)] Menu REF meof;
};
[association] class OfferS {
  Speciality REF speof;
  [Min (1)] CharacteristicRestaurant REF charof;
};

```

Fig. 8. MOF description of the class diagram in Figure 7.

Interval Constraints For Computer Graphics

Plots, Tessellations And Animations

Belaid Moa

Department of Computer Science,
University of Victoria, Victoria, Canada

bmoa@cs.uvic.ca,

WWW home page: <http://www.cs.uvic.ca/~bmoa/>

Abstract. Interval constraints programming was used for plotting relations with only two variables. In this paper, we extend this idea to plot any kind of relations in 2 or 3 dimensions. This paper also discusses two applications of interval constraints, namely tessellation and animation. More specifically, the paper looks at triangular tessellation and then provides a framework for doing any kind of tessellation. In addition, the paper discusses different ways of doing correct animations. New software for plotting using interval constraints is described and used to show the power and the importance of interval constraints in computer graphics.

1 Introduction

Conventional plotting is based on sampling. Thus, its correctness and completeness is not guaranteed. Given an implicit function defined by $f(x, y) = 0$, one can plot y with respect to x by writing y as a function of x : $y = g(x)$, and then sampling g for certain values of x . In order to have a precise plotting of g , the rate of sampling has to be greater than twice the maximum frequency of g (this follows from Nyquist Sampling Criterion). In general, obtaining g from f is hard, and making the rate of sampling greater than twice the maximum change rate of g is even harder. Another way of plotting f is by using the square chains to approximate f [1]. This gives only an approximation of f and it supposes that f has implicit derivatives with respect to x and y . Numerical plotting algorithm described in [2] can also be used for plotting f . This algorithm, however, is limited to cases where f is a polynomial function with respect to x and y , and with rational numbers as coefficients.

As is the case for any computational result, the rounding errors make the outcome of plotting even worse. Thus, a curve plotted with conventional software packages cannot be guaranteed to be an accurate representation of the function being plotted.

With the invention of interval arithmetic, controlling rounding errors became possible. The papers that proposed the use of interval arithmetic for computer graphics include [15], [5], and [6].

Plotting the function f using interval arithmetic is based on an inclusion function F of f . Given a rectangle $X \times Y$, if 0 is in $F(X, Y)$, then we keep the rectangle $X \times Y$, otherwise we discard it. By using interval arithmetic, one can also plot relations

involving logical operators based on the inclusion relations. This works as long as the relations remain fixed, but if the relations start changing over time, as we shall see later, the use of interval arithmetic becomes tedious.

An alternative way of plotting is by means of interval constraints. According to our search, the only paper that discussed the use of interval constraints in computer graphics is [7]. Their system, called IA Solver, was used to plot relations with only two variables. This paper is an extension of their idea to handle 3D, tessellation and animation.

That is actually the main objective of writing this paper, that is to emphasize the importance of applying interval constraints in computer graphics.

In section 2, we present a brief background of interval constraints. An important idea presented in that section is the notion of constraint store. Section 3 shows how to plot any constraint store with respect to two or three selected variables. The usual way of plotting is by using a quadtree-like method that we like to refer to as rectangular tessellation. In section 4, we generalize this idea to do tessellation of any kind, especially triangular tessellation. The idea behind the process of doing tessellation leads us to the notion of animated/dynamic constraint stores. This can be used to do animations, which is discussed in section 5. In section 6, we briefly describe SHRINC.net Plotter Engine implemented recently in C#. Finally, we mention some problems that were encountered during our investigation and that can be explored in future work.

2 Interval Constraints

Interval constraints combines two wide areas in computer science: Constraint Processing and Interval Arithmetic.

Interval Arithmetic uses intervals to carry out safe numerical computations. An *interval* $[a, b]$ is the set of real numbers that are less than or equal to b , and greater than or equal to a . A *box* is a Cartesian product of some intervals. A *rectangle* is a Cartesian product of two intervals. All arithmetic operations can be extended into intervals as done in [12, 3, 4, 13]. As an example, the “+” operator can be extended as follows: $[a, b] + [c, d] = [a + c, b + d]$. For an interval X , $lb(X)$ denotes its lower bound, $ub(X)$ its upper bound, and $w(X)$ its width (i.e $w(x) = ub(X) - lb(X)$). For clarity, in the rest of the paper, we will represent variables using lower case letters and intervals using upper case letters.

Constraint Processing deals with solving problems that involve constraints. A *constraint* is a relation that restrains the values of certain variables [8]. Each variable belongs to a domain. In what follows, the domains considered are intervals. If x is a variable, then X denotes its domain.

A constraint is in general represented by an expression involving arithmetic and logic operators. A *constraint store* is a conjunction of several constraints. At each step of processing, a constraint store can expand with the addition of another constraint or shrink due to deletion of another constraint.

At any instant T , if a constraint store S has C_1, \dots, C_n for constraints, then its state is the Cartesian product of the domains of the variables involved in $C_1 \dots C_n$. If the intervals X_1, \dots, X_n are the domains of the variables involved in $C_1 \dots C_n$, then the state of S at T is denoted by $s(T) = X_1 \times \dots \times X_n$. The configuration

of S at T is the tuple $(\{C_1, \dots, C_n\}, s(T), s(0))$, where $s(0)$ is the initial state of S . The constraint store S is said to be *corrupted* at T iff the set difference $s(T)/s(0)$ contains a tuple that satisfies all the constraints in S . In other words, if at an instant T , the constraint store loses some values of the variables that satisfy all the constraints, then, it is corrupted. The significance of corrupt constraint stores boils down to the completeness requirement: any processing that corrupts a constraint store should be avoided.

To add a constraint C to S we use $S.add(C)$, to delete a constraint C' from S we use $S.delete(C')$, to save the current state of S we use $S.save()$, and to load the latest saved state we use $S.load()$. The constraints of a constraint store S_1 can all be added to S by using $S.merge(S_1)$. As a result of merging, the domains of the variables in S and S_1 are set to the values they have in S_1 . The constraints that are common to S_1 and S can be deleted from S by using $S.separate(S_1)$. The state of S can be forced to a certain value s by using $S.setState(s)$.

A state of S can be reduced to another state without corrupting S by repeatedly using operators called *domain reduction operators*, DROs for short. These operators are associated with each constraint, and are either *primitive* or *composite*. The primitive DROs are efficiently computable and are, in general, associated with primitive constraints. For instance, the constraint $\text{Sum}(x,y,z)$ defined by $x + y = z$ has the following DRO:

$$X' = X \cap (Z - Y)$$

$$Y' = Y \cap (Z - X)$$

$$Z' = Z \cap (X + Y)$$

where the intervals X , Y and Z are the old domains of x , y and z respectively, and The intervals X' , Y' and Z' are their new domains.

A composite DRO is an operator associated with a complex constraint, and is, in general, built by decomposing the composite constraint into primitive constraints, and then applying the propagation algorithm (shown in Figure 1) based on the DROs of the primitive constraints.

```

put all constraints into the active set A
while ( A ≠ ∅ ) {
    choose a constraint C from A
    apply the DRO associated with C
    if one of the domains has become empty, then stop
    add to A all constraints involving variables whose domains have changed, if any
    remove C from A
}

```

Fig. 1. Propagation algorithm.

Consider the composite constraint defined by $x^2 + y^2 = 1$. This constraint can be decomposed into the following primitive constraints: $u = x^2$; $v = y^2$; $u + v = 1$. Now

that we have the primitive constraints, we can easily compute the composite DRO of the constraint $x^2 + y^2 = 1$ by applying the propagation algorithm to $u = x^2; v = y^2; u + v = 1$.

Applying propagation algorithm to a constraint store S means applying propagation algorithm to the constraints involved in S . S has a *solution* iff there is a tuple in $s(0)$ that satisfies all the constraints of S .

A constraint store S can be checked for consistency by calling the function $S.isConsistent()$. This function returns true iff none of the domains of the variables become empty when the propagation algorithm is applied to S . It is worth mentioning that $S.isConsistent()$ is true does not necessarily mean that S has a solution. It only means that inconsistency cannot be proved by using the propagation algorithm.

In general, the propagation algorithm is not enough to get sufficient results. To prove that the current state has some part that can be discarded, one needs to *split*. *Splitting* a constraint store S means producing a set of constraint stores that are similar to S except that the union of their initial states has to be equal to the current state of S . For example, if we have a constraint store with the following configuration ($\{x^2 = y, x^2 + y^2 = 1\}, [-1, 1] \times [-1, 1], [-2, 2] \times [-2, 2]$), then by using splitting we could produce the following constraint stores:

$$\begin{aligned} & (\{x^2 = y, x^2 + y^2 = 1\}, [-1, 1] \times [-1, 0], [-1, 1] \times [-1, 0]), \\ & (\{x^2 = y, x^2 + y^2 = 1\}, [-1, 1] \times [0, 1/2], [-1, 1] \times [0, 1/2]), \\ & (\{x^2 = y, x^2 + y^2 = 1\}, [-1, 1] \times [1/2, 1], [-1, 1] \times [1/2, 1]) \end{aligned}$$

If S has a solution, then it is in one of the constraint stores. Hence, an advantage of using splitting is that we simply discard subsets of the state that are proved not to contain a solution, without corrupting the constraint store.

3 Plotting Relations Correctly

In [7], the authors showed how to plot relations of two variables. Their way of plotting can be generalized to plot a relation with any number of variables (of course greater than or equal to 2) and in 2 or 3 dimensional space (2D or 3D).

3.1 Heterogeneous plotting of relations in 2D

Let us consider as example the constraint store S containing the constraint $C(x, y, z)$ defined by the expression $z = x^2 + y^2$ with a state $s(0) = X \times Y \times Z = [-1, 1] \times [-1, 1] \times [0.25, 0.5]$. The DRO of this composite constraint is computed by applying the propagation algorithm to the primitive constraints: $u = x^2, v = y^2$, and $z = u + v$. These constraints have primitive DROs that can be implemented easily in any programming language e.g. *C#*. Suppose we choose x and y as the variables with respect to which we want to plot $z = x^2 + y^2$. Even though this relation involves three variables, it can be plotted as easily as plotting a relation with two variables. Figure 2 shows the plotting of S with respect to x and y .

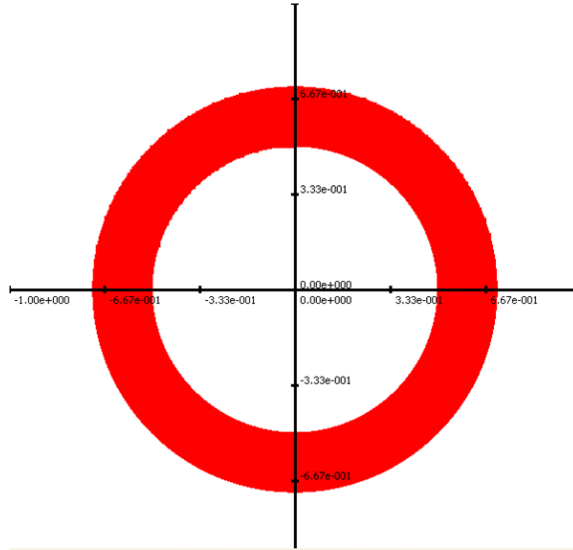


Fig. 2. Curve representing $z = x^2 + y^2$.

The conventional plotting packages cannot plot such a relation, and even if they can, they employ twisted techniques to accomplish this.

An algorithm that plots any kind of relation, or in general, any constraint store S , in $2D$ is as shown in Figure 3.

This algorithm looks similar to the one given in [7] except that instead of being able to plot a relation with two variables, it is able to plot a relations with several variables ($r(x, y)$ in [7] is replaced by S). A tiny technical difficulty that wasn't discussed in [7] is what should be done when splitting a rectangle $X \times Y$ for which X is small enough and Y is not. This problem can be solved by bisecting only Y . Hence, instead of four constraint stores only two will be produced. An algorithm that deals with this difficulty is shown in Figure 4.

3.2 Heterogeneous Plotting of relations in 3D

Suppose we have the constraint store with the unique constraint $x^2 + y^2 + z^2 \leq 1$ with an initial state $s(0) = X \times Y \times Z = [-1, 1] \times [-1, 1] \times [-1, 1]$. To plot such a constraint store in $3D$, we use the algorithm shown in Figure 5.

This algorithm is basically an extension of the algorithm in Figure 3 to three dimensions.

3.3 Homogeneous Plotting

The algorithms in Figure 3, Figure 4 and Figure 5 were qualified heterogeneous because they plot the whole S_i before plotting S_{i+1} . This behavior causes the plotting to spend

```

void plot2D(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ ) {
  if (! $S$ .isConsistent()){
    paint the rectangle  $X \times Y$  white;
    return;
  }
  if ( $X \times Y$  is small enough) {
    paint the rectangle  $X \times Y$  red;
    return;
  }
  split  $S$  into  $S_1, S_2, S_3, S_4$  by bisecting  $X$  and  $Y$ ;
  plot2D( $S_1, x, y$ ); plot2D( $S_2, x, y$ ); plot2D( $S_3, x, y$ ); plot2D( $S_4, x, y$ );
}

```

Fig. 3. Heterogeneous plotting algorithm in 2D.

```

void plot2D(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ ) {
  if (! $S$ .isConsistent()){
    paint the rectangle  $X \times Y$  white;
    return;
  }
  if ( $X \times Y$  is small enough) {
    paint the rectangle  $X \times Y$  red;
    return;
  }
  if ( $X$  and  $Y$  are not small enough){
    split  $S$  into  $S_1, S_2, S_3, S_4$  by bisecting  $X$  and  $Y$ ;
    plot2D( $S_1, x, y$ ); plot2D( $S_2, x, y$ ); plot2D( $S_3, x, y$ ); plot2D( $S_4, x, y$ );
    return;
  }
  if ( $X$  is not small enough){
    split  $S$  into  $S_{X1}$  and  $S_{X2}$  by bisecting  $X$ ;
    plot2D( $S_{X1}, x, y$ ); plot2D( $S_{X2}, x, y$ );
    return;
  }
  if ( $Y$  is not small enough){
    split  $S$  into  $S_{Y1}$  and  $S_{Y2}$  by bisecting  $Y$ ;
    plot2D( $S_{Y1}, x, y$ ); plot2D( $S_{Y2}, x, y$ );
    return;
  }
}

```

Fig. 4. Improved Heterogeneous plotting algorithm in 2D.

```

void plot3D(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ , Variable  $z$ ) {
  if (! $S$ .isConsistent()){
    paint the rectangle  $X \times Y \times Z$  white;
    return;
  }
  if ( $X \times Y \times Z$  is small enough) {
    paint the rectangle  $X \times Y \times Z$  red;
    return;
  }
  split  $S$  into  $S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8$  by bisecting  $X, Y$  and  $Z$ ;
  plot3D( $S_1, x, y, z$ ); plot3D( $S_2, x, y, z$ ); plot3D( $S_3, x, y, z$ ); plot3D( $S_4, x, y, z$ );
  plot3D( $S_5, x, y, z$ ); plot3D( $S_6, x, y, z$ ); plot3D( $S_7, x, y, z$ ); plot3D( $S_8, x, y, z$ );
}

```

Fig. 5. Heterogeneous plotting algorithm in $2D$.

too much time in one box before going to the other boxes. Sometimes this way of plotting is not suitable since the user may want to see the shape of the plot after a few computations. Moreover, the user may want to stop the plotting at will and to resume it later on. For these reasons we propose the following algorithm in which we use a queue to keep track of the boxes that still need to be processed. The queue, referred to as RedBoxes, is initialized to contain the initial state of S . At each step, the box B at the front is removed from the queue, and is used to check the consistency of S . If S is consistent and the box B is not small enough, then we split B and add its parts to the end of the queue. The algorithm stops once the queue is empty. The algorithm is shown in Figure 6.

The algorithm in Figure 6 can be extended easily to $3D$.

For the sake of clarity, I provided Figure 7 that shows how the algorithm in Figure 4 differs from the algorithm in Figure 6.

4 Tessellation

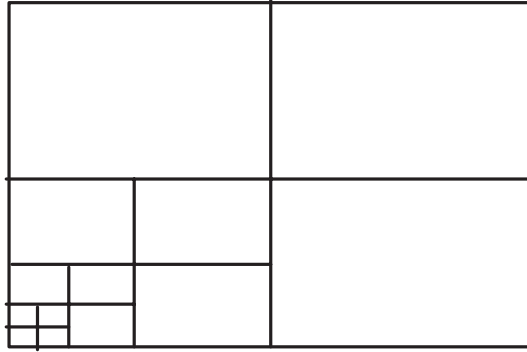
A regular tiling of polygons (in two dimensions), polyhedra (three dimensions), or polytopes (n dimensions) is usually referred to as a tessellation. To put it simply, a tessellation is a repeating pattern of interlocking shapes. For example, the algorithms for plotting in $2D$ split a box, with respect to x and y , into four rectangles, each of which is again split into four rectangles. This process continues until the boxes are small enough or proved inconsistent. In other words, our algorithms tessellate the curve using rectangles. This kind of tessellation is referred to as *rectangle tessellation*. This type of tessellation is not all the time the good way of approximating a curve. OpenGL, for instance, tessellates using triangles. In what follows, we will show that such a tessellation can be easily done using interval constraints. Before embarking into details, the reader may find Figure 8 helpful to see the difference between rectangular tessellation and triangular tessellation.

```

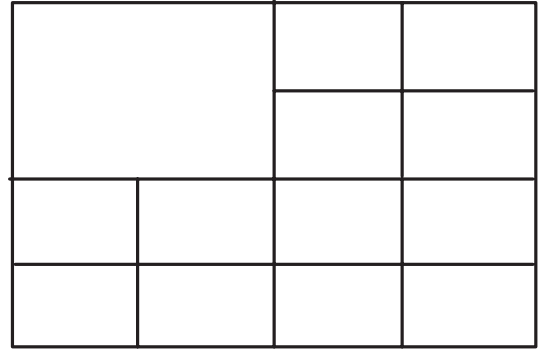
void plot2D(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ ) {
  initialize RedBoxes to  $s(0)$ ;
  while (RedTess is not empty and the plotting not stopped){
    Box  $b$  = RedBoxes.Remove();
     $S$ .setState( $b$ );
    if ( ! $S$ .isConsistent() ) {
      paint  $X \times Y$  white;
      return;
    }
    if ( $X \times Y$  is small enough) {
      paint the rectangle  $X \times Y$  red;
      return;
    }
    if ( $X$  and  $Y$  are not small enough){
      split  $b$  into four boxes bisecting both  $x$  and  $y$ ;
      add the boxes to the queue;
      continue;
    }
    if ( $X$  is not small enough){
      split  $b$  into two boxes bisecting  $X$ ;
      add the boxes to the queue;
      continue;
    }
    if ( $Y$  is not small enough){
      split  $b$  into two boxes bisecting  $Y$ ;
      add the boxes to the queue;
    }
  }
}

```

Fig. 6. Homogeneous plotting algorithm in 2D.

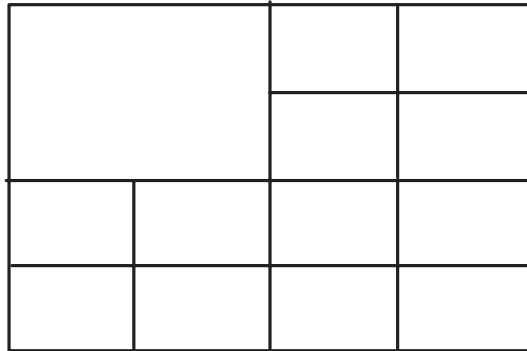


(a)

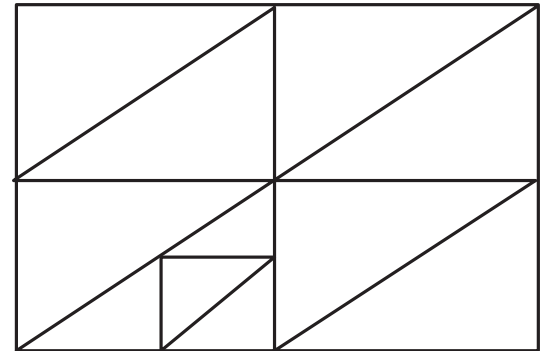


(b)

Fig. 7. (a) The order of splitting of the algorithm in Figure 3. (b) The order of splitting of the algorithm in Figure 6.



(a)



(b)

Fig. 8. (a) Rectangular tessellation. (b) Triangular tessellation.

In the algorithms of the previous section, changing the state of a constraint store was sufficient to do rectangle tessellation. For triangular tessellation this is not enough; new constraints need to be added to the constraint store. To check the consistency of the constraint store S on the triangle T shown in Figure 9, we set domains of x and y to X and Y respectively, and we add the constraint $y \leq w(Y)/w(X)(x - lb(X)) + lb(Y)$ to the constraint store. The triangle T can be viewed as a constraint store containing the constraint $y \leq w(Y)/w(X)(x - lb(X)) + lb(Y)$ with initial state set to $X \times Y$. With this view, the consistency of the constraint store S can be checked by merging S and T . Instead of keeping track of the boxes using RedBoxes queue, we keep track of the triangles that still need to be processed. We use the queue RedTriangles for this purpose.

The consistency of T' can be checked in a similar manner.

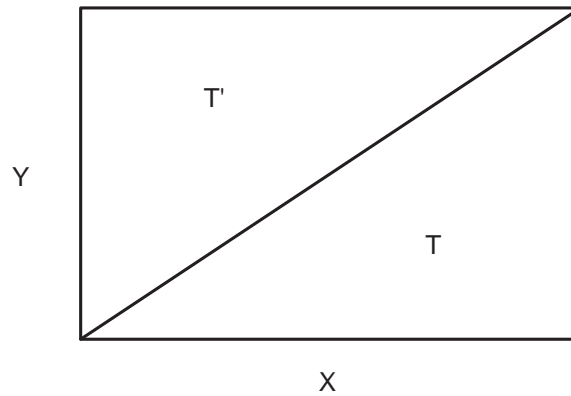


Fig. 9. Triangular tessellation.

It is worth noting that once $S.isConsistent()$ returns, we delete the added new constraints and load its old state. This allows the constraint store to be intact and ready for further use. For the triangular tessellation, the algorithm of plotting becomes the one shown in Figure 10.

The triangular tessellation approach can be generalized to perform any kind of tessellation that the user might want to. The algorithm in Figure 11 is a generalized version of the algorithm given in Figure 10. This algorithm uses the function $Tess()$ that takes an area and tessellates it. The $Tess()$ function returns a set of constraint stores each of which represents a tessellated area.

Before proceeding to the next section, it is useful to mention that a constraint store may be plotted faster with triangular tessellation than rectangular tessellation. However, the opposite could also be true. Therefore, the user has a choice to apply whichever tessellation he wants depending on the type of curve that needs to be plotted.

```

void plot2D(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ ) {
    split the rectangle  $X \times Y$  into two triangles  $T$  and  $T'$ ;
    add  $T$  and  $T'$  to RedTriangles;
    while(RedTriangles is not empty and the plotting not stopped) {
        Constraint Store  $T = \text{RedTriangle.front}()$ ;
         $S.\text{save}()$ ;
         $S.\text{merge}(T)$ ;
        if (  $!S.\text{isConsistent}()$  ) {
            paint  $T$  white;
             $S.\text{separate}(T)$ ;
             $S.\text{load}()$ ;
            return;
        }
        if ( $T$  is small enough) {
            paint  $T$  red;
            continue;
        }
        split  $T$  into four triangles;
        add these triangles to the RedTriangles queue;
    }
}

```

Fig. 10. Plotting using triangular tessellation.

```

void plot2D(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ ) {
    tessellate the rectangle  $X \times Y$  using Tess();
    add the constraint stores returned by Tess() to the queue RedTess;
    while(RedTess is not empty and the plotting not stopped) {
        Constraint Store  $S' = \text{RedTess.front}()$ ;
         $S.\text{save}()$ ;
         $S.\text{merge}(S')$ ;
        if (  $!S.\text{isConsistent}()$  ) {
            paint  $T$  white;
             $S.\text{separate}(S')$ ;
             $S.\text{load}()$ ;
            return;
        }
        if ( $S'$  is small enough) {
            paint  $S'$  red;
            continue;
        }
        split  $S'$  using Tess();
        add the constraint stores returned by Tess() to the RedTess queue;
    }
}

```

Fig. 11. A generalized algorithm for plotting using any kind of tessellation.

5 Animation

We encountered two ways of changing the constraint store: one is by changing its state and the other is by changing its constraints. These changes allow the constraint store to change its configuration. By carefully controlling the instants in time where the constraint store changes, one can have an animated constraint store. This animation can be visualized once the constraint store is plotted as done in the next subsection.

5.1 Animated Plotting

All the algorithms mentioned so far, plot a constraint store as fast as the underlying hardware is capable of. For the purpose of animation, the user may want to slow down the process of plotting. To do that, we propose the algorithm shown in Figure 12. This algorithm only draws a predetermined number of rectangles N that exist in the Red-Boxes queue. This algorithm alone is not enough to plot a constraint store. That is why it is called within a timer as shown in Figure 13. The frequency of the timer and the predetermined number of rectangles N determine the animation of the constraint store.

Figure 14 shows the process of animation when plotting a circle. The timer is set to 100 msec, and N to 4.

The figure shows the animation done by SHRINC.net to plot a circle. Snapshot (a) shows the initial state of animation when the circle is imprecise. The animation proceeds by moving from (a) to (b), and along each step, the circle is redrawn more sharply. The animation ends when the circle becomes totally sharp and clear, as shown in snapshot (c). This might seem a bit trivial, but it can actually be used to simulate/animate a bacterial culture for example, where bacteria grow or die over time.

5.2 Animated Scene

In the previous subsection, we showed how to do animated plotting by changing only the state of a constraint store. In this subsection we will present an example in which the animation is done by changing the constraints in a constraint store. The example that is considered is the shape of the sun during sunset. The sun is represented as a circle and the earth as an ellipse. Initially, the constraint store is set to contain only the constraint representing the sun. At each tick of the timer the constraint “a part of the sun is hidden behind the earth” is added to the store.

Formally, let us suppose that the equation representing the sun in the middle of the day is $x^2 + y^2 \leq 1$, and the equation of the earth is $x^2/4 + (y + 4)^2 \leq 1$. Initially, the constraint $x^2 + y^2 \leq 1$ is added to the constraint store. At each tick of the timer, the center of the earth is moved by δy and we add the constraint $x^2/4 + (y + 4 - \delta y)^2 > 1$ to the constraint store and then we plot it.

Figure 15 shows the animation produced by such a process. There are two main advantages of this process: one is that it provides more depth and granularity to the picture being drawn/animated, and the other advantage is that the animation is guaranteed to be an exact simulation of the phenomenon being simulated. This is really crucial and beneficial to areas of engineering such as aircraft simulation, rocket simulation etc., since precision and accuracy are the main concerns in these types of simulations. This cannot

```

void animatedPlotting(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ , int  $N$ ) {
    int  $k = 0$ ;
    while (RedBoxes is not Empty and  $k$  is less than  $N$ ){
         $k++$ ;
        Box  $b =$  RedBoxes.Remove();
         $S$ .setState( $b$ );
        if ( ! $S$ .isConsistent() ) {
            paint  $X \times Y$  white;
            return;
        }
        if ( $X \times Y$  is small enough) {
            paint the rectangle  $X \times Y$  red;
            return;
        }
        if ( $X$  and  $Y$  are not small enough){
            split  $b$  into four boxes bisecting both  $x$  and  $y$ ;
            add the boxes to the queue;
            continue;
        }
        if ( $X$  is not small enough){
            split  $b$  into two boxes bisecting  $X$ ;
            add the boxes to the queue;
            continue;
        }
        if ( $Y$  is not small enough){
            split  $b$  into two boxes bisecting  $Y$ ;
            add the boxes to the queue;
        }
    }
}

```

Fig. 12. Algorithm used for animated plotting.

```

void atTimerTick(Constraint Store  $S$ , Variable  $x$ , Variable  $y$ , int  $N$ ) {
    while (the plotting is not stopped){
        animatedPlotting( $S, x, y, N$ );
    }
}

```

Fig. 13. Function called at each tick of the timer.

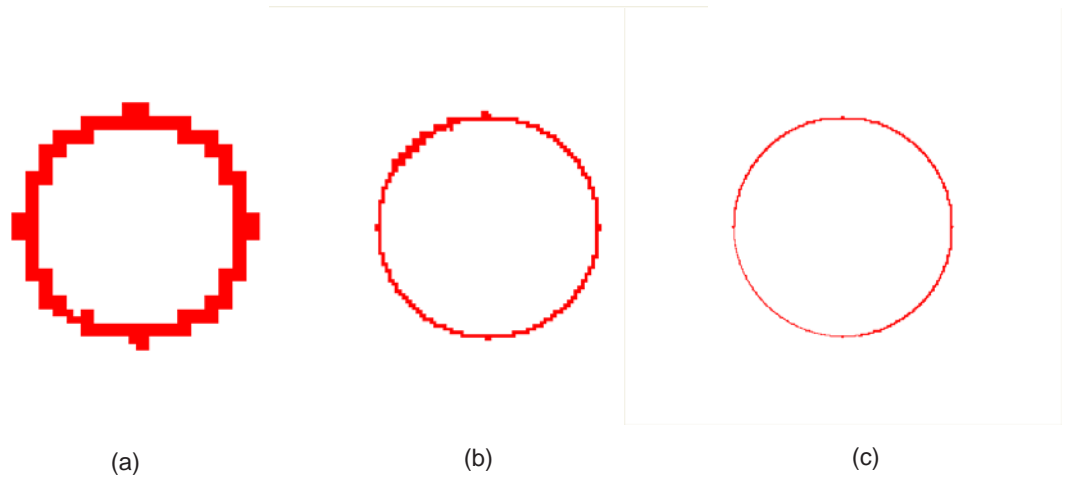


Fig. 14. Animated Circle Plotting.

be achieved with traditional methods of animation since they are error-prone because of rounding, and cannot produce an exact depiction/simulation of the phenomenon under consideration.

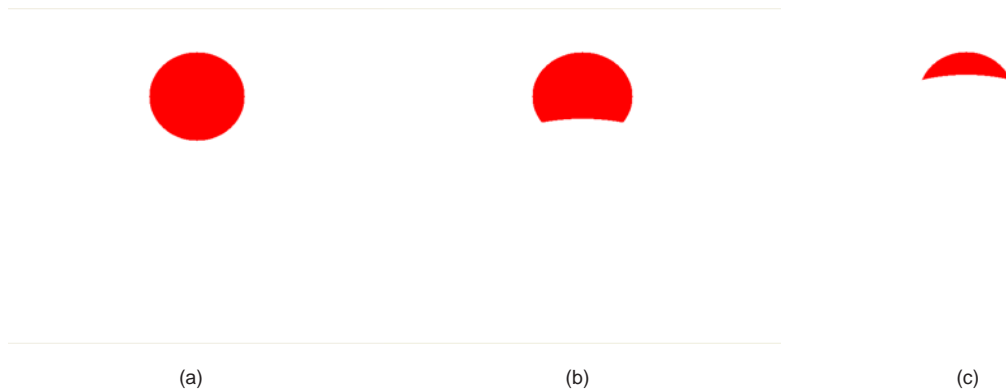


Fig. 15. Sunset animation.

Collisions of two balls, a bouncing ball on a wall, and other animations can be easily implemented in a similar way.

It is useful to point out that for efficiency purposes, we associated with each constraint store a stack that keeps track of the set of the rectangles painted Red. In this way,

when a new constraint is added to the store, we don't have to start the plotting all over again. Instead, we only check the consistency of the rectangles stored in the stack.

6 SHRINC.net Plotter Engine

SHRINC is open software managed by M.H. van Emden at the University of Victoria. SHRINC is originally developed using *C* and *C++*. Recently, we implemented SHRINC in .net using *C#*. Thus the name SHRINC.net. We used software rounding instead of the hardware rounding for portability issues. We also developed an engine for interval arithmetic calculation, an engine for Optimization, an engine for solving constraint systems and an engine for Plotting. The graphical interface of SHRINC.net is shown in Figure 16.

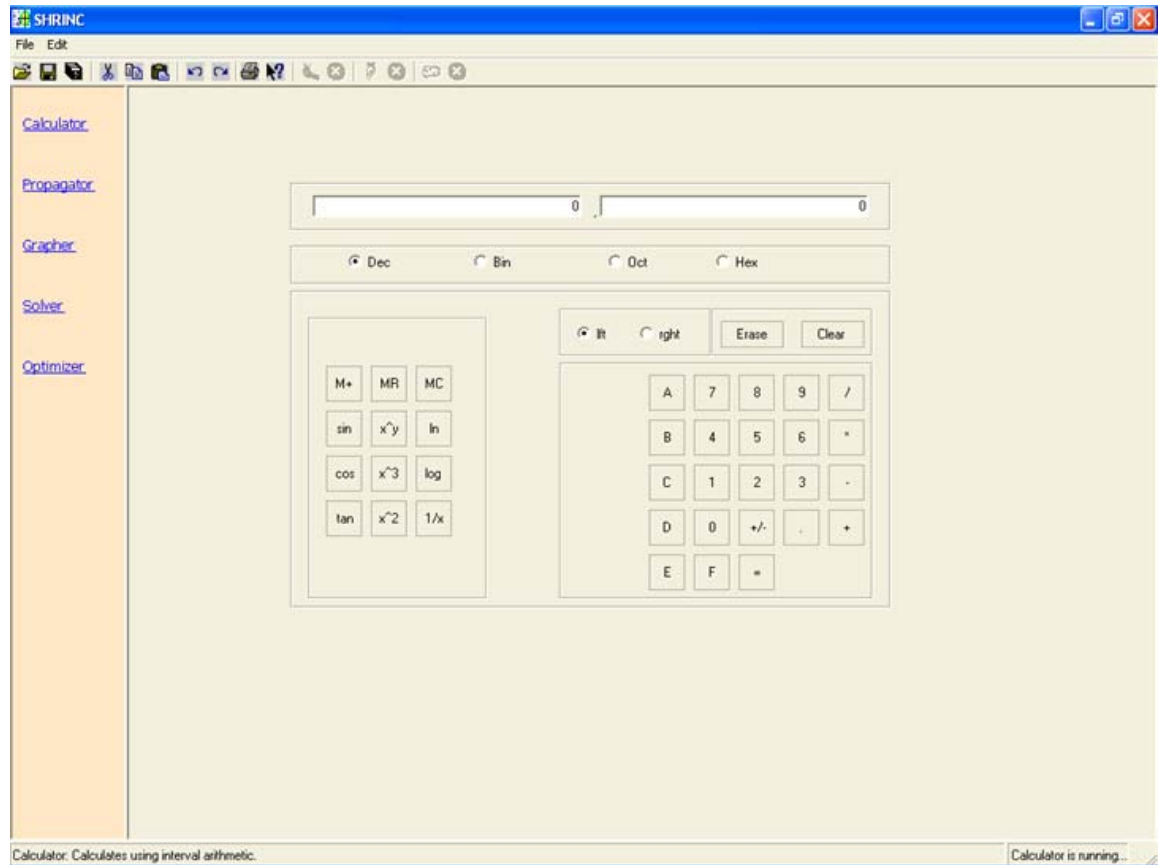


Fig. 16. SHRINC .net GUI.

To plot any constraint store using SHRINC .net Plotter, first write the expressions of the constraints separated by semicolons in the constraint store, specify the domains of the variables, choose the dimension of the space you would like to plot in, and finally provide the axes of your plot.

The steps are shown in figures 17, 18, and 19.

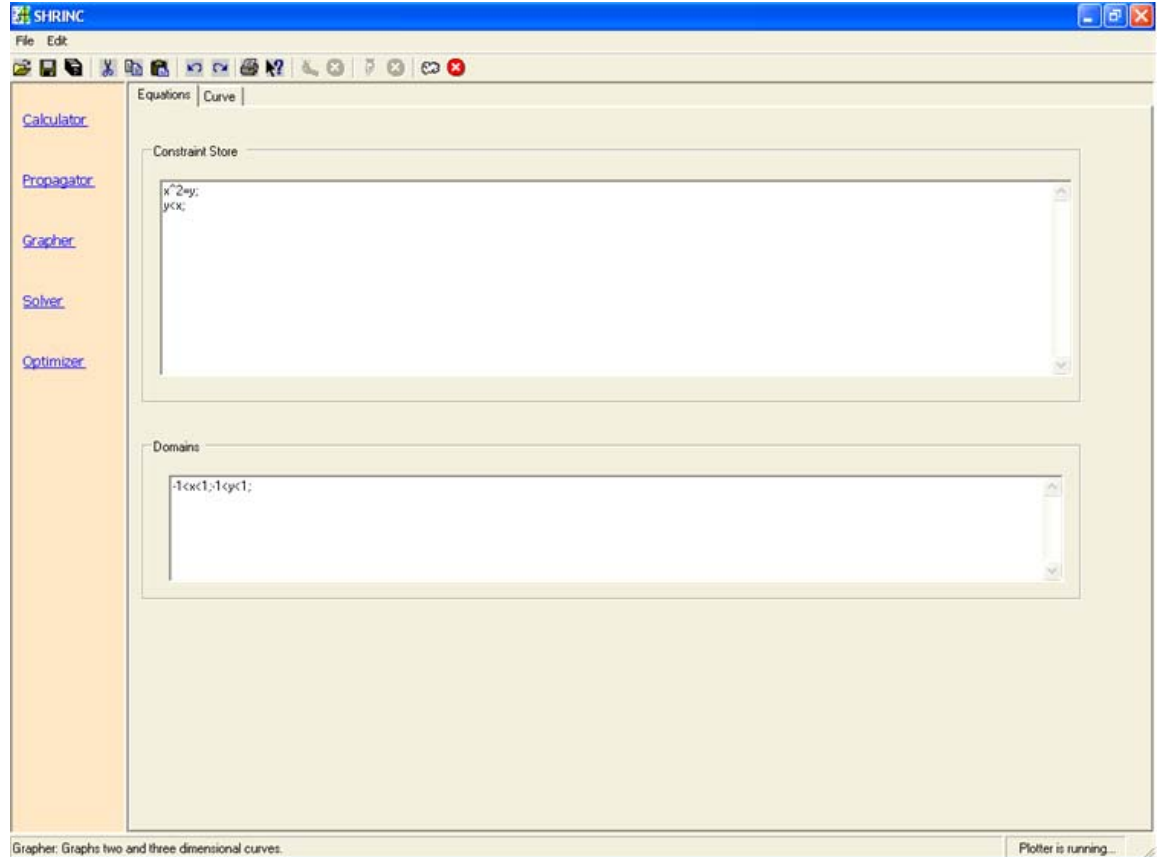


Fig. 17. Writing the equations of the constraints .

Before concluding it is worth mentioning that for 3D plotting under SHRINC .net we used an OpenGL library for C#.

7 Conclusion

The main objective of this paper is to emphasize the importance of interval constraints in computer graphics. Such importance becomes a must when the user is interested in the

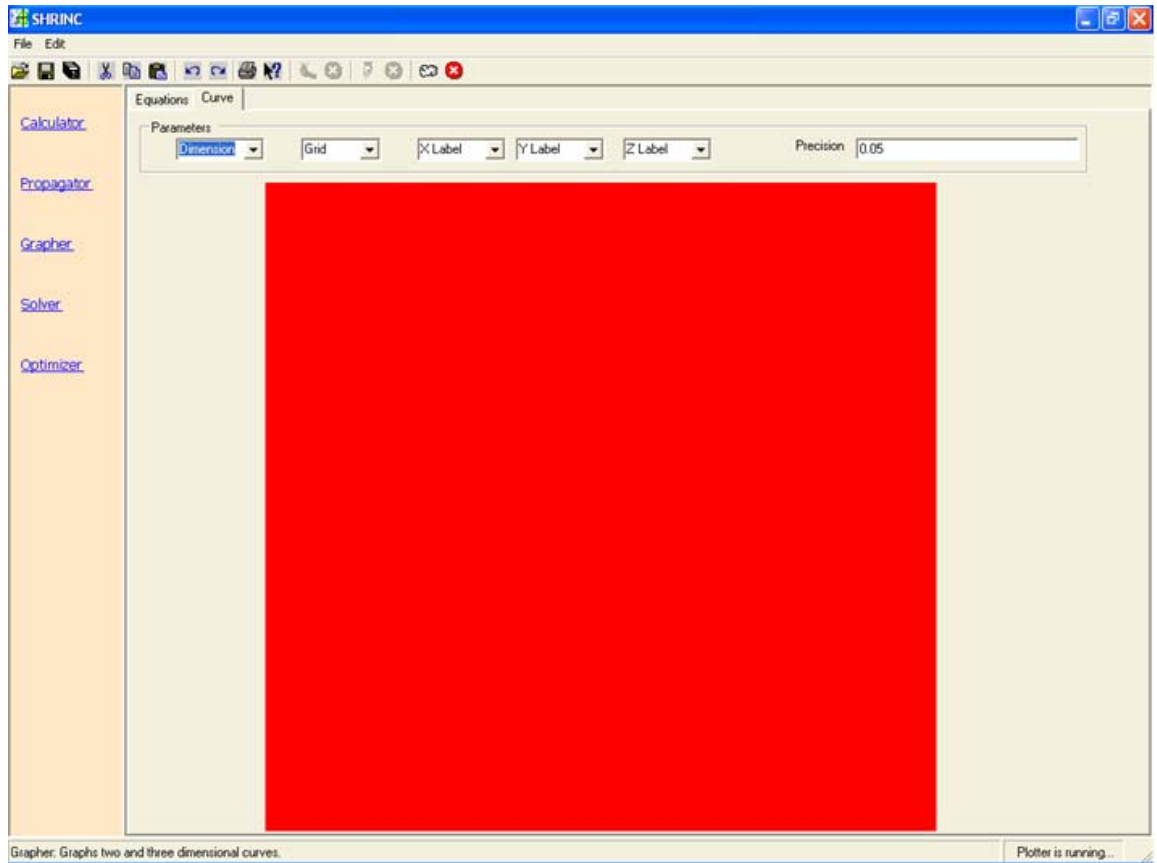


Fig. 18. Use the Tab to go to the plotting area.

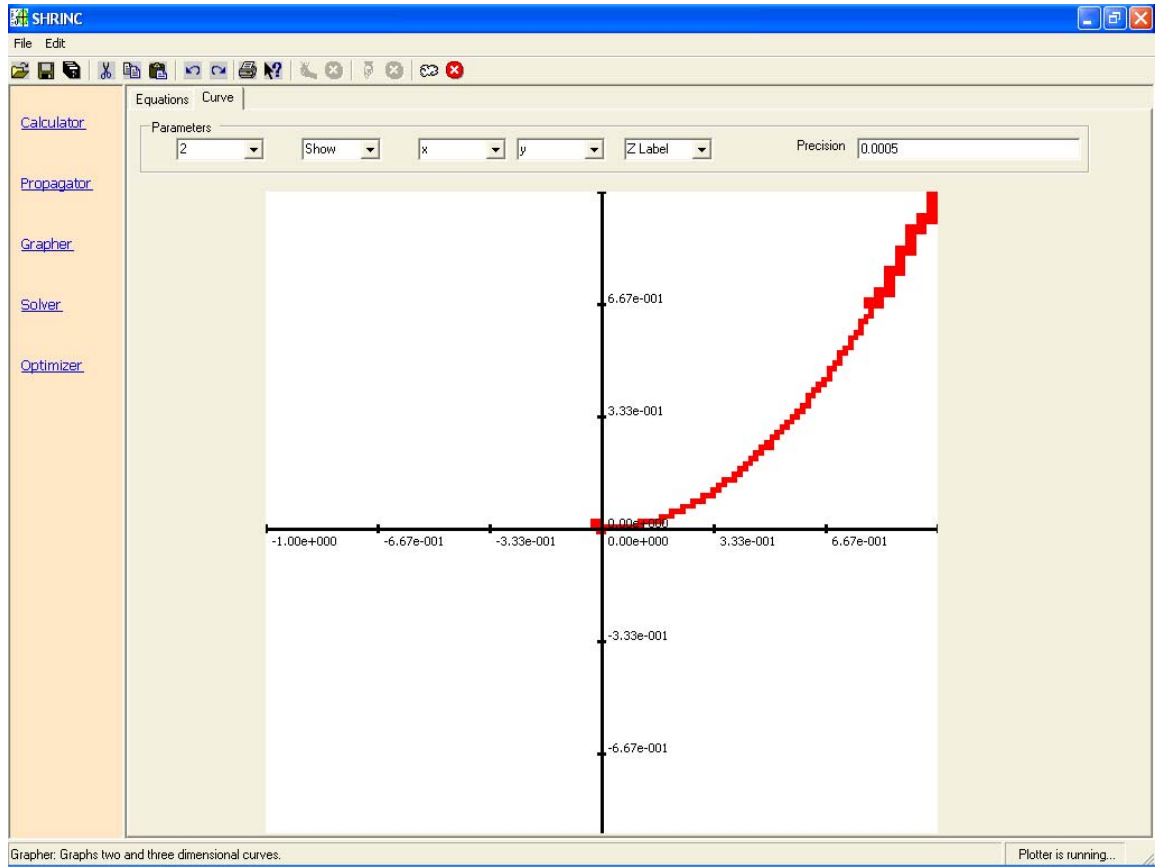


Fig. 19. Choose the parameters of your plotting and press the plotting button on the toolbar.

correctness and the completeness of plotting, tessellating and animating. Conventional plotting can only give an approximation of the actual plotting. Is this enough a reason to leave the conventional plotting packages? The answer of course is no. Plotting using interval constraints suffers from being too slow when the constraint store has many constraints. Plotting a relation with long expression is time consuming. An idea that can be exploited in future work is to use iterative widening [11] to plot large constraint stores correctly based on conventional plotting packages.

The efficiency problem is even severe when doing animation using interval constraints; when the frequency of the timer is small, the scene starts flickering. Moreover, due to the randomness of the propagation algorithm, the time it takes for `isConsistent()` to return is random. An interesting alternative is to use other consistency techniques instead of decomposing the constraint into primitives. Consistency techniques such as Box-Consistency and Bound-Consistency may yield better results than the approach we followed [9, 10, 14].

An interesting difficulty we faced when doing 3D is that OpenGL library used does not allow the user to plot a specific region. The whole scene has to be redrawn each time a box is proved inconsistent. Redrawing the whole scene causes a severe flickering and slows the plotting process. An alternative solution that we are investigating right now is to use DirectX instead.

In short, according to our findings, interval constraints are a much better approach to graphics if the priority is to acquire correctness and completeness. Their downside however is that they are slow. But this will obviously change in the future as the technology becomes available. In the meantime, one solution is to integrate the interval constraints technique with the conventional methods so as to produce results that are not only accurate but also fairly fast.

Acknowledgments

I would like to thank the paper reviewers for their comments; Dr. Maarten van Emden for his support and advice; and N. Akif for his suggestions.

References

1. Ephraim Cohen. A method for plotting curves defined by implicit equations. In *Journal of Computer Graphics*, volume 10, number 2, pages 263–265, 1976.
2. Dennis S. Arnon. Topologically reliable display of algebraic curves. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques.*, pages 219–227, ACM Press, 1983 ,
3. T. Hickey, Q. Ju, and M.H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5). Sept. 2001.
4. T.J. Hickey, M.H. van Emden, and H. Wu. A unified framework for interval constraints and interval arithmetic. In Michael Maher and Jean-François Puget, editors, *Principles and Practice of Constraint Programming – CP98*, pages 250 – 264. Springer-Verlag, 1998. Lecture Notes in Computer Science 1520.
5. Jeff Tupper. Reliable Two-Dimensional Graphing Methods for Mathematical Formulae with Two Free Variables. In *SIGGRAPH 2001 Conference Proceedings.*, August 2001 ,

6. Tom Duff. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques.*, pages 131–138, ACM Press, 1992 ,
7. T. Hickey, Q. Ju, and M.H. van Emden. Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. *Reliable Computing*, volume 6, pages 81–92, 2000.
8. Roman Bartak. Theory and practice of constraint propagation. In *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control (CPDC 2001)*, pages 7–14. J. Figwer(Editor), 2001.
9. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proc. 1994 International Symposium*, pages 124–138, 1994.
10. Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising hull and box consistency. In *Proceedings of the 16th International Conference on Logic Programming*, pages 230–244. MIT Press, 1999.
11. Pascal Van Hentenryck and Laurent Michel and Yves Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.
12. Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
13. Arnold Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.
14. Laurent Granvilliers and Frric Goualard and Frric Benhamou Box Consistency through Weak Box Consistency. In *ICTAI*, pages 373-380, 1999.
15. John M. Snyder. *Interval analysis for computer graphics*. In *Journal of Computer Graphics*, volume 26, number 2, pages 121–130, 1992.

Multiplex Dispensation Order Generation for Pyrosequencing

Mats Carlsson¹ and Nicolas Beldiceanu^{2,3}

¹ SICS, P.O. Box 1263, SE-164 29 KISTA, Sweden
matsc@sics.se

² LINA FRE CNRS 2729
École des Mines de Nantes
FR-44307 NANTES Cedex 3, France
Nicolas.Beldiceanu@emn.fr

³ This research was carried out while N. Beldiceanu was at SICS.

Abstract. This article introduces the multiplex dispensation order generation problem, a real-life combinatorial problem that arises in the context of analyzing large numbers of short to medium length DNA sequences. The problem is modeled as a constraint optimization problem (COP). We present the COP, its constraint programming formulation, and a custom search procedure. We give some experimental data supporting our design decisions. One of the lessons learnt from this study is that the ease with which the relevant constraints are expressed can be a crucial factor in making design decisions in the COP model.

1 Introduction

This article introduces the multiplex dispensation order generation problem, a real-life combinatorial problem that arises in the context of analyzing large numbers of short to medium length DNA sequences with the Pyrosequencing method [1–3]. DNA, or deoxyribonucleic acid, is the molecule encoding the genetic information of all cellular life. It is a double-stranded polymer formed from four nucleotides (bases) abbreviated A, C, G, T.

The Pyrosequencing method was developed based on the combination of four enzymes: (i) polymerase (bacterial), (ii) sulphurylase (yeast), (iii) luciferase (firefly) and (iv) apyrase (potato). The idea is to use a single, *input* strand of the DNA to be analyzed as a template for synthesizing the complementary, *output* strand. The synthesis proceeds by incorporating one nucleotide at a time, the incorporation of a specific nucleotide being quantitatively detectable as visible light; see Fig. 1.

Assuming a sample from a monoploid species, i.e. individuals have a single copy of each gene, each cycle of the method proceeds as follows¹: A specific nucleotide is dispensed, i.e. added to the reaction, and if it matches the current base of the input strand, it is incorporated into the output strand, and the next input base becomes current. If it matches a stretch of k bases, it is incorporated k times, and the k^{th} next input base

¹ We ignore details of the chemistry and the issue of complementary DNA bases.

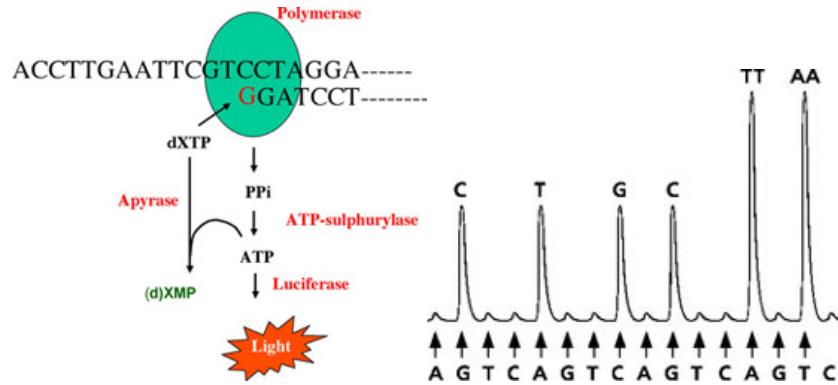


Fig. 1. The Pyrosequencing method. A sample is analyzed wrt. a cyclic dispensation order AGTCAGTC... Assuming a monoploid sample, the sequence CTGCTTAA is obtained.

becomes current. Each incorporation event is accompanied by emission of visible light with energy proportional to the amount of incorporated nucleotide, i.e. to k , the number of matched bases. No matter whether the nucleotide was incorporated, the enzymes ensure that any residue of it is degraded, and the reaction is ready for the next cycle. A histogram with one peak of height k per cycle captures the outcome of the reaction.

Fig. 1 suggests that the nucleotides should be dispensed in a cyclic order AGTCAGTC... However, in a typical application such as forensic analysis, most of the sequence is known in advance, and only single positions or small stretches are variable. It is these *polymorphic* parts that are of interest, i.e. that carry signal. For example, in single nucleotides polymorphism (SNP) analysis, a sequence has a single variable position. In our work, a sequence is allowed to have more general variable parts and is described by a *template*. A template is simply a restricted form of regular expression where n denotes a single nucleotide ($n \in \{A, C, G, T\}$), xy denotes concatenation, x/y denotes alternatives, ϵ denotes the empty string, and $[x]$ is the same as x/ϵ , assuming that x and y are themselves templates. Parentheses are used for grouping. An *allele* of a template s is a string over the nucleotides described by s .

While a cyclic dispensation order is appropriate for completely unknown sequences, it is very wasteful in terms of reagents and throughput for typical templates. Hence, an algorithm that can generate a custom, *simplex* dispensation order (SDO) for a specific template is of great practical value. Such an algorithm was the subject matter of a previous paper [4].

In many applications of the method, the DNA sample is from a diploid species, i.e. one where each individual has inherited one copy of each gene from the father and one from the mother. This means that the sample contains a multiset of two alleles, which may or may not be distinct. If they are distinct, the histogram obtained will be equivalent to superimposing the histograms for the two alleles. The requirement that an SDO should allow for determining the genotype of the input sample is a major challenge for the SDO generation algorithm [4], but is not discussed further in this article. Ex. 1 shows an input template and a dispensation order computed by the SDO generation

algorithm. Such dispensation orders drive the analysis equipment, which can perform some number (typically 96) of reactions in parallel.

Example 1. Suppose that we are given the following input template, describing a piece of DNA from a diploid species:

CA(A/C)[AGA][TG][A/G]TATTC

The template contains 4 polymorphisms generating 24 alleles, shown in Fig. 2. A dispensation order for this template is:

CACAGATGATATC

Template	CA	A/C	[AGA]	[TG]	[A/G]	TATTC
Allele 1	CA	A	ϵ	ϵ	ϵ	TATTC
Allele 2	CA	A	ϵ	ϵ	A	TATTC
Allele 3	CA	A	ϵ	ϵ	G	TATTC
Allele 4	CA	A	ϵ	TG	ϵ	TATTC
Allele 5	CA	A	ϵ	TG	A	TATTC
Allele 6	CA	A	ϵ	TG	G	TATTC
Allele 7	CA	A	AGA	ϵ	ϵ	TATTC
Allele 8	CA	A	AGA	ϵ	A	TATTC
Allele 9	CA	A	AGA	ϵ	G	TATTC
Allele 10	CA	A	AGA	TG	ϵ	TATTC
Allele 11	CA	A	AGA	TG	A	TATTC
Allele 12	CA	A	AGA	TG	G	TATTC
Allele 13	CA	C	ϵ	ϵ	ϵ	TATTC
Allele 14	CA	C	ϵ	ϵ	A	TATTC
Allele 15	CA	C	ϵ	ϵ	G	TATTC
Allele 16	CA	C	ϵ	TG	ϵ	TATTC
Allele 17	CA	C	ϵ	TG	A	TATTC
Allele 18	CA	C	ϵ	TG	G	TATTC
Allele 19	CA	C	AGA	ϵ	ϵ	TATTC
Allele 20	CA	C	AGA	ϵ	A	TATTC
Allele 21	CA	C	AGA	ϵ	G	TATTC
Allele 22	CA	C	AGA	TG	ϵ	TATTC
Allele 23	CA	C	AGA	TG	A	TATTC
Allele 24	CA	C	AGA	TG	G	TATTC

Fig. 2. Alleles of the template used in Ex. 1

Suppose that the analysis of a DNA sample from an individual of this species using this dispensation order yields the histogram shown in Fig. 3. From a case analysis of the possible combinations, we can deduce the genotype of the individual, namely:

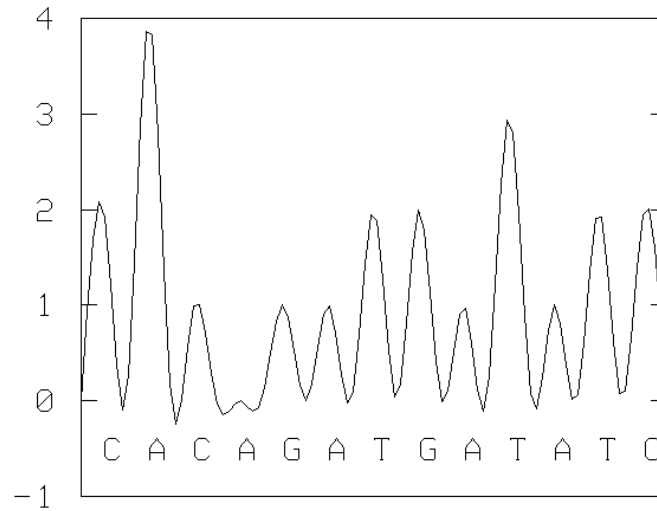


Fig. 3. Histogram obtained in Ex. 1

Polymorphism	Parent 1	Parent 2
A/C	A	C
AGA/ε	AGA	ε
TG/ε	TG	ε
A/G/ε	G	ε

and the putative sequences (there are two feasible allele pairs, 12+13 and 7+18):

Multiset 1	Multiset 2
CAAAGATGGTATTC	CAAAGATATTC
CACTATTC	CACTGGTATTC

Note that, for each polymorphism, we can only deduce the multiset of variants present in the sample, i.e. we cannot unambiguously determine which parent contributed which variant. For the practical applications such as SNP analysis, it suffices to determine the multisets. \square

In this article, we take another step: in a typical laboratory setting, there is a large number of samples and different templates to analyze. If templates are different enough, the amount of parallelism can be increased by multiplexing some reactions. This is done by placing the samples in the same well, and analyzing the mixture with a *multiplex*

dispensation order (MDO). The point is that the length of an MDO can be much shorter than the total length of its constituent SDOs. Moreover, an MDO does not normally span the entire SDOs: it suffices to span the polymorphic parts plus a little more, to be made precise in Sect. 2.

A problem instance is defined by (i) a number of SDOs, and (ii) for each SDO, some first and follow facts computed by the SDO generation algorithm; these are explained in Sect. 3.

A dispensation order (SDO or MDO) is a sequence of *items*, each defined by (i) its nucleotide, (ii) its *minimal multiplicity* (the minimal height of the peak that this item in the sequence will yield), and (iii) its *maximal multiplicity* (the maximal height of the peak that this item in the sequence will yield). If the minimal and maximal multiplicities are equal, we say that we have a *fixed item*, otherwise we have a *variable item*.

For yielding a correct analysis result, it is crucial that the height of every peak corresponding to a variable item be precisely determined. Due to details of the sample preparation procedures, the concentration of DNA may vary a lot from sample to sample. This means that the signal strength, i.e. the height of each peak, is only comparable *within* a single (simplex) reaction, and not *across* reactions. This also means that there is no universal height unit; peak heights are always relative within one reaction. In order to calibrate the peak heights, the MDO generation algorithm must ensure to include at least one (preferably more than one) item of fixed and low multiplicity, so called *norm-items*, from each SDO.

Two SDO items can be *coalesced* into a single MDO item if they have the same nucleotide. That is, the peaks that the SDO items would yield become superimposed in the peak yielded by the coalesced MDO item. The more items are coalesced, the shorter the resulting MDO becomes. Fixed items may be coalesced arbitrarily. No two variable items can ever be coalesced, as doing so would confuse two signals. A variable item should preferably not be coalesced with a fixed item, as doing so compromises the accuracy of measuring its peak height. The COP model allows it, at a penalty.

Example 2. In this example, we show three input templates, their corresponding SDOs, and finally how these SDOs align into an optimal MDO. The resulting MDO will yield 6 peaks corresponding to variable items. The height of these peaks will allow for determining the exact allele composition of the input sample. The MDO also contains 7 norm-items out of 10 fixed items, yielding 10 peaks of fixed height. 3 of the MDO items were the result of coalescing SDO items. Finally, the MDO did not have to span to the end of any of the SDOs.

Template	SDO
TCC (C/T) GGGAAATAATC	TCTGATATC
AT (C/T) CAGGGGGTGCTT	ATCAGTGCT
GCTTCA (A/G) TGGA	GCTCAGTGA

SDO 1	TCT	G	A	T:ATC
SDO 2		ATCAG	T	G :CT
SDO 3			GCT	C AGT:GA
MDO	TCTATCAGCTACGAGT			
Items	nvvnvvnfnfnnnvfvf			

Legend:

- v Variable item.
- n Fixed item, norm-item.
- f Fixed item, not a norm-item.
- : Cut-off position (L in the constraint optimization model)

□

The rest of the paper is organized as follows: in Sect. 2, the MDO problem is defined in more details; in Sect. 3, we present a COP model; in Sect. 4, we describe a custom search procedure; and in Sect. 5, we show a brief experimental evaluation of some design choices. Finally, we conclude.

2 Multiplex Dispensation Orders

An MDO is optimal if it fulfills Requirements 1–5 and has minimal cost according to Requirements 6–8. Some of these requirements, and their details, may seem arbitrary, but their purpose is to reduce the risk of an incorrect interpretation of the resulting histogram.

Definition 1. *An item yields a norm-item if and only if it is fixed, of multiplicity at most 3, and non-coalesced.* □

Requirement 1. *The MDO must span at least 4 fixed items of each SDO.* □

Requirement 2. *The MDO must span at least 1 fixed item after the last variable item of each SDO.* □

Requirement 3. *The MDO must span at least 1 norm-item of multiplicity 1 of each SDO.* □

Requirement 4. *The MDO must span at least 1 norm-item within 10 dispensations from each variable item.* □

Requirement 5. *No variable item may be coalesced with another variable item.* □

Requirement 6. *For each SDO, the MDO spanning less than 2 norm-items incurs a penalty of 3.* □

Requirement 7. *A fixed item coalesced with a variable item incurs a penalty of 3.* □

Requirement 8. *The MDO incurs a penalty equal to its length.* □

3 A Constraint Optimization Model

3.1 Introduction

In this section, we will develop a COP model of the multiplex dispensation order generation problem. In the rest of this section, we will describe SDOs in more detail, some parameters which can be derived from the SDOs, the constraint variables, the cost function, and the constraints. Finally, we will describe the search procedure.

3.2 Problem Parameters

We need the following notation. (s, i) denotes an item with attributes:

- s , the SDO in which it occurs
- i , the position in s at which it occurs,
- $\text{nuc}(s, i)$, its nucleotide,
- $\text{min}(s, i) \geq 0$, its minimal multiplicity,
- $\text{max}(s, i) > 0$, its maximal multiplicity,
- $\text{fix}(s, i)$ holds if and only if $\text{min}(s, i) = \text{max}(s, i)$.
- $\text{first}(s, i)$ holds if and only if there is an allele of s such that (s, i) is the first item yielding a nonzero peak.

For items (s, i) and (s, j) , we have:

- $\text{follow}(s, i, j)$ holds if and only if there is an allele of s such that (s, j) is the first item yielding a nonzero peak after item (s, i) .
- $\text{nuc}(s, i) \neq \text{nuc}(s, j)$ if $\text{follow}(s, i, j)$ holds.

3.3 Problem Variables

Each position in the MDO is called a *cycle*, numbered from 1 upward. For each SDO, a cycle has to be assigned to each item from the leftmost position up to the earliest cut-off point L admitted by the constraints.

- $F \geq 0$ is the value of the cost function.
- $L \geq 1$ is the last dispensation cycle.
- $C_{(s,i)} \geq 1$ is the cycle that is assigned to item (s, i) .
- $D_k^n \geq 1$ is the cycle in which the k^{th} dispensation of nucleotide $n \in \{A, C, G, T\}$ occurs. For convenience, we assume that $D_0^n = 0$. The solution, i.e. the actual MDO, can be obtained by sorting the D_k^n by ascending value, and reading the nucleotide sequence.
- $E_{(s,i)(t,j)}$ is 1 if (s, i) and (t, j) are coalesced, i.e. $C_{(s,i)} = C_{(t,j)}$, and 0 otherwise, where $s \neq t$.
- $N_{(s,i)}$ is 1 if item (s, i) yields a norm-peak, and 0 otherwise (see below).

3.4 Cost Function

The cost function captures Requirements 6–8.

$$L + \sum \{3E_{(s,i)(t,j)} \mid \neg \text{fix}(s,i) \vee \neg \text{fix}(t,j)\} + \sum_s \begin{cases} 3, & \text{if } \sum_i \{N_{(s,i)}\} = 1 \\ 0, & \text{otherwise} \end{cases}$$

3.5 Problem Constraints

Constraints intrinsic to Pyrosequencing.

1. Only items with the same nucleotide can be coalesced.

$$\text{nuc}(s,i) \neq \text{nuc}(t,j) \Rightarrow C_{(s,i)} \neq C_{(t,j)} \quad (1)$$

2. The items $1, \dots, n$ of each SDO must be assigned in ascending order:

$$C_{(s,1)} < C_{(s,2)} < \dots, \text{ for each SDO } s \quad (2)$$

3. A $\text{first}(s,j)$ fact expresses the constraint that $C_{(s,j)}$ be the very first dispensation of $\text{nuc}(s,j)$. Similarly, a $\text{follow}(s,i,j)$ fact expresses the constraint that $C_{(s,j)}$ be the first dispensation of $\text{nuc}(s,j)$ after cycle $C_{(s,i)}$. These constraints model the Pyrosequencing reaction, prevent invalid multiplex dispensation orders, and form the link between the $C_{(s,i)}$ and the D_k^n variables.

$$\text{first}(s,j) \Rightarrow C_{(s,j)} = D_1^{\text{nuc}(s,j)} \quad (3)$$

$$\text{follow}(s,i,j) \Rightarrow \exists k \mid D_{k-1}^{\text{nuc}(s,j)} < C_{(s,i)} < C_{(s,j)} = D_k^{\text{nuc}(s,j)} \quad (4)$$

4. A nucleotide cannot be dispensed twice in a row. (It could, but it would be pointless, as the second dispensation would always yield a zero peak.)

$$D_1^n + 2 \leq D_2^n \wedge D_2^n + 2 \leq D_3^n \wedge \dots, \text{ for each } n \in \{A, C, G, T\} \quad (5)$$

5. All dispensations occur in distinct cycles.

$$\text{all_distinct}(\{D_k^n\}) \quad (6)$$

6. Dispensing past the end of any SDO would cause the reaction to fall off the piece of DNA being analyzed, yielding nonsensical output.

$$L \leq C_{(s,i)}, \text{ for each SDO } s \text{ with final item } i \quad (7)$$

Definition of norm-item.

$$N_{(s,i)} = \begin{cases} 1, & \text{if } C_{(s,i)} \leq L \wedge \text{fix}(s,i) \wedge \max(s,i) \leq 3 \wedge \sum_{t,j} E_{(s,i)(t,j)} = 0 \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

Requirement 1.

$$L \geq C_{(s,i)}, \text{ for each SDO } s \text{ with 4th fixed item } i \quad (9)$$

Requirement 2.

$$L > C_{(s,i)}, \text{ for each SDO } s \text{ with last variable item } i \quad (10)$$

Requirement 3.

$$\exists i \mid N_{(s,i)} = 1 \wedge \max(s,i) = 1, \text{ for each SDO } s \quad (11)$$

Requirement 4.

$$\exists i \mid N_{(s,i)} = 1 \wedge |C_{(s,i)} - C_{(s,j)}| \leq 10, \text{ for each variable item } (s,j) \quad (12)$$

Requirement 5. The following two constraints are equivalent, but posting both of them improves constraint propagation:

$$\text{all_distinct}(\{C_{(s,i)} \mid \neg \text{fix}(s,i)\}) \quad (13)$$

$$\sum \{E_{(s,i)(t,j)} \mid \neg \text{fix}(s,i) \wedge \neg \text{fix}(t,j)\} = 0 \quad (14)$$

3.6 An Earlier Model

Our choice of problem variables is far from obvious, and was not the first one that we came up with. The $C_{(s,i)}$ variables are indispensable, as they are crucial for several of the constraints. The D_k^n variables, however, are not: the solution, i.e. the actual MDO, can be obtained by sorting the $C_{(s,i)}$ by ascending value, removing duplicates, and reading the nucleotide sequence. However, it is awkward at best to express the intrinsic constraints shown in (3) and (4) in terms of $C_{(s,i)}$ variables only.

In an earlier model, we had:

- D_k^n variables replaced by $K_c \in \{A, C, G, T\}$ representing the nucleotide to dispense in the c^{th} cycle, directly encoding the solution.
- (3–6) replaced by:

$$K_{C_{(s,i)}} = \text{nuc}(s,i), \text{ for each item } (s,i) \quad (15)$$

$$\text{first}(s,j) \Rightarrow (\forall p \geq 1 : p < C_{(s,j)} \Rightarrow K_p \neq \text{nuc}(s,j)) \quad (16)$$

$$\text{follow}(s,i,j) \Rightarrow (\forall p \geq 1 : C_{(s,i)} \leq p < C_{(s,j)} \Rightarrow K_p \neq \text{nuc}(s,j)) \quad (17)$$

$$\forall i \geq 1 : K_i \neq K_{i+1} \quad (18)$$

The K_c variables have much smaller domains than the D_k^n variables, and so a model using the former would suggest a much smaller search space in theory. However, with this formulation too, (16) and (17) are somewhat awkward to express and would probably require to be implemented as custom global constraints for efficiency. This fact prompted us to come up with the model using D_k^n variables.

3.7 Implementation Details

The model was implemented using the CLP(FD) library of SICStus Prolog [5]. Most of the constraints in Sect. 3.5 have straightforward CLP(FD) formulations, but the following points are worth noting:

1. Reified arithmetic constraints come in very handy in linking the $E_{(s,i)(t,j)}$ and $N_{(s,i)}$ variables. A reified constraint has the form $c \equiv b$, where c is any constraint and b is a 0/1 variable. It expresses the fact that the value of b (1 for true, 0 for false) reflects the truth value of c .
2. An “at least one” constraint (existential quantifiers in (11) and (12)) is conveniently expressed using an `element/3` constraint. Suppose `Cs` is the list of $C_{(s,i)}$ variables for given s , and `Xs` is a corresponding list of 0/1 variables, each reflecting the value $N_{(s,i)} = 1 \wedge \max(s, i) = 1$. Then the two CLP(FD) constraints:

```
element(I, Xs, 1),
element(I, Cs, Ci)
```

hold with `Ci = C(s,i)` if there is a norm-item (s, i) satisfying Requirement 3.

3. Similarly, the existential quantifier in (4) is expressed as follows, where `Ci` is $C_{(s,i)}$, `Cj` is $C_{(s,j)}$, and `Ds` is the list of $D_k^{\text{nuc}(s,j)}$ variables. Note the use of `[-1 | Ds]` in the third line, which allows for accessing the domain variable $D_{k-1}^{\text{nuc}(s,j)}$ using the index $K = k$. Note also that the condition $C_{(s,i)} < C_{(s,j)}$ has already been captured by (2), and so is not re-expressed here:

```
Ch #< Ci,
element(K, Ds, Cj),
element(K, [-1 | Ds], Ch)
```

4 Search Procedure

We now present the main features of our search procedure, beginning with the task of finding feasible solutions, and moving on to the task of finding optimal solutions. In Sect. 5, we evaluate our design decisions wrt. some experimental data.

4.1 Feasible Solutions

As usual in constraint programming, the search for a feasible solution is done by depth-first search: each node (choicepoint) in the search tree corresponds to a partially constructed MDO where a specific nucleotide n is being chosen. Each daughter node corresponds to a specific choice of n . Whenever the constraint solver detects an infeasibility, the current branch is abandoned and the search backtracks to the most recent choicepoint. We now describe the specifics of the search.

Variable choice. We use a cycle-based variable order, building the MDO sequentially from left to right, and stopping when a feasible value for L has been reached. As no problem variables directly correspond to the MDO, we instead assign the D_k^n variables, in each cycle advancing one of the D^n arrays. The relevant $C_{(s,i)}$ variables get assigned by constraint propagation. In other words, the assignment effectively simulates the reaction, which in reality proceeds from left to right.

Value choice. In each cycle we have a choice between at most four nucleotides. The choice is guided by looking ahead in the SDOs. Let i'_s and i''_s be the next two unassigned items in SDO s . For each nucleotide $n \in \{A, C, G, T\}$ we compute:

$$R_1(n) = \sum_s \begin{cases} 1, & \text{if } \text{nuc}(s, i'_s) = n \\ 0, & \text{otherwise} \end{cases}$$

$$R_2(n) = \sum_s \begin{cases} 1, & \text{if } \text{nuc}(s, i''_s) = n \\ 0, & \text{otherwise} \end{cases}$$

If $R_1(n) > 0$, then n is a candidate nucleotide for the next cycle. The candidates are tried in descending order of $R_1(n) - R_2(n)$, using a *least recently used* rule to break ties.

4.2 Optimal Solutions

Let “feasible” be a procedure implementing the search for a feasible solution, with or without a cost bound. It is assumed to compute the cost of that solution. The standard constraint programming approach to optimization consists in the iterated search for feasible solutions with successively tighter and tighter cost bound; see Alg. 1.² We used this scheme with two modifications, detailed below.

Binary search. Instead of successively tightening the cost bound, we narrowed the feasible range of the cost in a binary search fashion until it becomes fixed; see Alg. 2. This was found to sometimes lead to fewer iterations and faster convergence.

Oracles. Since our variable choice is static and our value choice is independent of the cost function and its bounds, we can follow the approach of Van Hentenryck and Le Provost [6]. In each iteration of binary search with a tighter upper bound, we use an *oracle*, or computation path. The oracle mentions, for each cycle, the nucleotide that was assigned to that cycle in the best solution so far, and any untried choices. The oracle remains valid as long as we follow its advice. On backtracking to an untried choice, the oracle is no longer valid. The advantage of using oracles is twofold:

- The oracle guides the search so as to avoid parts of the search space that have already been exhausted.
- As long as we follow the oracle’s advice, i.e. up to backtracking, we avoid the cost of computing a value order for the current cycle.

² The expression $\min(F)$ yields the current cost lower bound.

The pseudocode for our full search procedure is shown in Alg. 3. Here, the procedure “feasible” has been augmented with an input oracle O , which remains valid while the leftmost branch is taken in all choices. Also, “feasible” is assumed to compute an output oracle encoding the part of the search space that is still unexplored.

```

PROCEDURE label( $F$ ) : (false, true)
Ensure:  $F$  is an output parameter.
1: if feasible = true then
2:    $F \leftarrow$  the current cost value
3:   display data about the solution found
4:   return true
5: else
6:   return false
7: end if
PROCEDURE opt( $F$ ) : (false, true)
Ensure:  $F$  is an output parameter, receiving the optimal cost value on success.
1: if label( $F_0$ ) = false then
2:   return false
3: end if
4: while min( $F$ ) <  $F_0$  do
5:   if post  $F < F_0 \wedge$  label( $F_1$ ) = true then
6:      $F_0 \leftarrow F_1$ 
7:   else
8:      $F \leftarrow F_0$ 
9:   end if
10: end while
11: return true

```

Algorithm 1: Optimization

5 Performance Evaluation

In this section, we provide some experimental data supporting our design choices. The data is presented in four scatter plots (see Fig. 4), each comparing the implemented, *baseline* solution with a *mutated* version in which some feature has been changed or disabled. Each scatter plots shows some points and an $x = y$ line. The X and Y coordinates of each point give the runtime in milliseconds of a given problem instance for the baseline and mutated version, respectively.

In the evaluation, we used 184 random test instances, each consisting of three multiplexed templates, generated as follows:

- Each test instance consisted of three templates to be multiplexed.
- Each template was 12 nucleotides long and contained at least one polymorphic nucleotide. Nucleotides were polymorphic with probability 1/12.
- Each test instance was feasible and its optimal solution was found in less than one minute.

```

PROCEDURE labelbin( $F$ ) : (false, true)
Ensure:  $F$  is an output parameter.
1: if feasible = true then
2:    $F \leftarrow$  the current cost value
3:   display data about the solution found
4:   return true
5: else
6:   return false
7: end if
PROCEDURE optbin( $F$ ) : (false, true)
Ensure:  $F$  is an output parameter, receiving the optimal cost value on success.
1: if labelbin( $F_0$ ) = false then
2:   return false
3: end if
4: while  $\min(F) < F_0$  do
5:   if post  $F \leq (\min(F) + F_0)/2 \wedge$  labelbin( $F_1$ ) = true then
6:      $F_0 \leftarrow F_1$ 
7:   else
8:     post  $F > (\min(F) + F_0)/2$ 
9:   end if
10: end while
11: return true

```

Algorithm 2: Optimization with binary search

```

PROCEDURE labelbinora( $O_0, F, O$ ) : (false, true)
Ensure:  $F$  and  $O$  are output parameters.
1: if feasible( $O_0$ ) = true then
2:    $F \leftarrow$  the current cost value
3:    $O \leftarrow$  the current oracle
4:   display data about the solution found
5:   return true
6: else
7:   return false
8: end if
PROCEDURE optbinora( $F$ ) : (false, true)
Ensure:  $F$  is an output parameter, receiving the optimal cost value on success.
1: if labelbinora( $\square, F_0, O_0$ ) = false then
2:   return false
3: end if
4: while  $\min(F) < F_0$  do
5:   if post  $F \leq (\min(F) + F_0)/2 \wedge$  labelbinora( $O_0, F_1, O_1$ ) = true then
6:      $F_0 \leftarrow F_1$ 
7:      $O_0 \leftarrow O_1$ 
8:   else
9:     post  $F > (\min(F) + F_0)/2$ 
10:   end if
11: end while
12: return true

```

Algorithm 3: Optimization with binary search and oracles

We now present the test results.

Plot no oracle. In this plot, the mutated version does not use oracles in the search. All problem instances were speeded up by using oracles, and the greatest speed-ups were observed for the more time-consuming instances.

Plot no binary search. In this plot, the mutated version uses an iterated search for feasible solutions with successively tighter and tighter cost bounds. The data shows that this approach is consistently better than the binary search version, although some real-life instances are counter-examples.

Plot one-step look-ahead. In this plot, the mutated version uses a value-choice heuristic that ignores the R_2 term. The R_2 term has no apparent effect on the test set, but was noted to sometimes make a difference on real-life instances supplied to us by Pyrosequencing AB.

Plot no LRU. In this plot, the mutated version has no LRU rule for breaking ties. It shows the same lack of effect as the previous plot, but the resulting MDOs on real-life instances looked better to human experts.

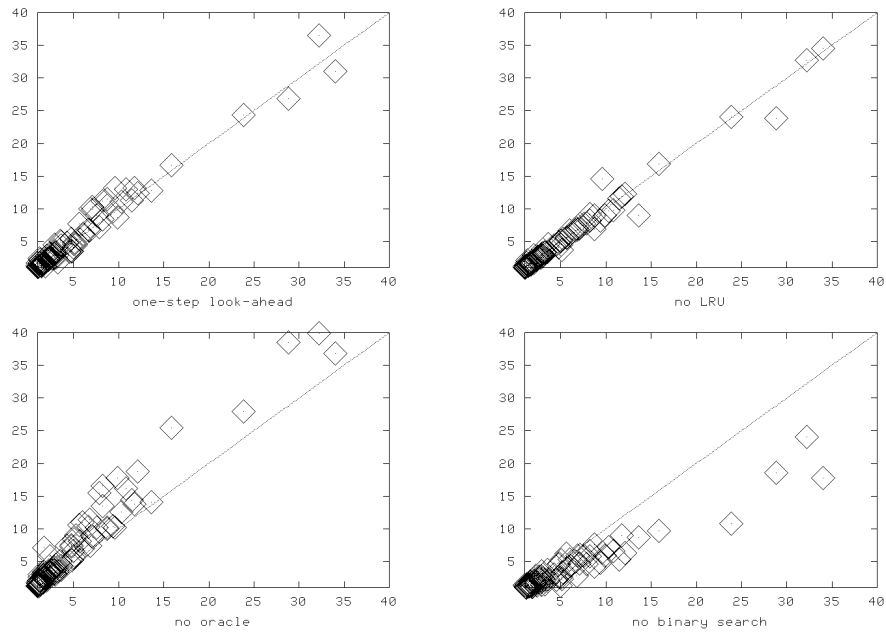


Fig. 4. Time in seconds to reach optimality for 184 random instances. Each plot compares the baseline version with a mutated one.

6 Conclusion

We have introduced the multiplex dispensation order generation problem, a real-life combinatorial problem that arises in the context of analyzing large numbers of short to medium length DNA sequences. We have modeled the MDO generation problem as a COP. The model has been implemented using the CLP(FD) library of SICStus Prolog [5], and we comment on the encoding of some of the constraints of the COP model. A crucial part of the implementation is a custom search procedure, which is described in detail. Finally, we provide some experimental data supporting the design choices of the search procedure.

One of the lessons learnt from this study is that different designs of the COP model (D_k^n vs. K_c variables) can differ a lot in the ease with which the relevant constraints are expressed. This was a crucial factor in the choice of the model that uses D_k^n variables, although these have much larger domains and so would suggest a larger search space in theory.

Another lesson learnt, or reinforced, is that “the devil is in the details” when it comes to search procedures—a combination of techniques was necessary in order to achieve good performance.

Acknowledgments

The research reported herein was funded by Pyrosequencing AB³. The graphics used in Fig. 1 is courtesy of the Division of Molecular Biotechnology, Royal Institute of Technology, Stockholm⁴. We are grateful to Per Mildner and the anonymous referees for their useful comments.

References

1. Nyrén, Pettersson, and Uhlén. Solid phase DNA minisequencing by an enzymatic lumino-metric inorganic pyrophosphate detection assay. *Anal. Biochem.*, 208:171–175, 1993.
2. Ronaghi, Karamohamed, Pettersson, Uhlén, and Nyrén. Real-time DNA sequencing using detection of pyrophosphate release. *Anal. Biochem.*, 242:84–89, 1996.
3. Ronaghi, Uhlén, and Nyrén. A sequencing method based on real-time pyrophosphate. *Science*, 281:363–365, 1998.
4. Mats Carlsson and Nicolas Beldiceanu. Dispensation order generation for pyrosequencing. In Yi-Ping Phoebe Chen, editor, *Proc. APBC2004*, volume 29 of *Conferences in Research and Practice in Information technology*, Dunedin, New Zealand, 2004. Australian Computer Society.
5. M. Carlsson et al. *SICStus Prolog User’s Manual*. Swedish Institute of Computer Science, release 3 edition, 1995. ISBN 91-630-3648-7.
6. Pascal Van Hentenryck and Thierry Le Provost. Incremental Search in Constraint Logic Programming. *New Generation Computing*, 9:257–275, 1991.

³ www.pyrosequencing.com

⁴ www.biotech.kth.se/molbio

Boosting Constraint Satisfaction using Decision Trees ^{*}

Barry O’Sullivan, Alex Ferguson, and Eugene C. Freuder

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{b.osullivan|a.ferguson|e.freuder}@4c.ucc.ie

Abstract. Constraint satisfaction is becoming the paradigm of choice for solving many real-world problems. To date, most approaches to constraint satisfaction have focused on solving a problem using some form of backtrack search. Furthermore, the typical view is that a constraint satisfaction problem will be solved only once. However, in many real-world contexts, problems are solved repeatedly over time. Also, such problems often exhibit some structure. This motivates the application of some form of learning to improve the performance of search from previously discovered solutions. In this paper we present an approach that uses knowledge about known solutions to a problem to improve search. The approach we present is based on a combination of decision tree learning and constraint satisfaction. We demonstrate that significant improvements, almost an order-of-magnitude, in search effort can be achieved using this hybrid approach over traditional search. We also show that the space complexity using this approach is almost negligible. This work is of interest in domains such as product configuration, and interactive constraint solving in general where the system takes the initiative by asking questions.

Keywords: Constraint satisfaction, Decision tree learning, Interactive search.

1 Introduction

Constraint satisfaction has become an important paradigm in Artificial Intelligence over the last 30 years [11, 15]. Informally, a constraint satisfaction problem (CSP) is defined by a set of variables, each of which has a corresponding set of possible values called its domain, and the task is to find values for each variable from its domain so that a set of constraints are satisfied. Constraint satisfaction is applicable to a wide variety of problems arising in scheduling, design, configuration, machine vision, temporal reasoning and planning [20].

Most constraint satisfaction techniques are based on a combination of inference and search. The traditional problem addressed by the constraint satisfaction

^{*} This work has received support from Science Foundation Ireland (Grant Number 00/Pl.1/C075). Barry O’Sullivan is also supported by Enterprise Ireland under their Basic Research Grant Scheme (Grant Number SC/02/289).

community is a CSP that must be solved only once. However, in many real-world domains, CSPs are solved many times over, for example if the CSP represents an intensional specification of a set of products available from a commercial web-site. In such a scenario users present queries to a web-site, possibly responding to a set of questions presented by the system, representing a set of choices or desires, and the task is to present a solution that satisfies the query or to report that none exists. In this context, we can regard the query as a set of unary constraints that instantiate some subset of the variables in the CSP. The task of the constraint solver is to decide whether this set of unary constraints can be extended to a complete solution or not.

In real-world application contexts, such as the one outlined above, the underlying CSP often exhibits some structure in terms of its solution space. For example, there may be many solutions to the problem that are different only in terms of a small number of assignments. Specifically, many real-world problems exhibit some form of interchangeability amongst values [7]; i.e. we may be free to change one assignment in a solution, holding the others fixed, and still have a solution. It seems likely in such cases that learning methods can be employed to take advantage of this structure in order to boost the performance of basic constraint solving. It is this observation that underpins the work we present here.

In this paper we consider an interactive scenario where users make choices about some system-determined subset of the variables in a problem and seek a solution that completes it. For example, imagine a car sales situation where a buyer states that he wishes to purchase a car having alloy wheels, a driver's airbag and a metallic colour¹. The task of the sales assistant is to recommend solutions that satisfy these constraints.

We present an approach that first considers a relatively small number of solutions of the CSP; these can be found either by search or, as is more likely in a real-world context, from a catalogue or brochure containing examples of solutions that are available. Using this set of positive examples (solutions), and a randomly generated set of negative examples (non-solutions), a decision tree is induced. This decision tree is processed in a number of ways to give us a variable ordering for instantiating variables and a source of additional constraints that can be used to further refine a subsequent search phase in order to find a solution. Note that in some situations, subsequent search is unnecessary since it can be proven by the decision tree alone that a solution exists or does not exist. In terms of the overall result, we demonstrate that close to an order-of-magnitude reduction in search effort is consistently possible for highly structured problems.

It should be noted that the ordering of the variables is determined by the system in order to minimize the search effort required to respond to the query. Therefore, the approach we present is most useful where the system takes the initiative in an interactive context.

Therefore, the primary contribution of this paper is a novel integration of machine learning and constraint satisfaction techniques for boosting the performance of query answering systems in domains that are representative of many

¹ We choose car configuration as our running example motivated by [1].

real-world problem classes such as product configuration [1, 17]. This integration, while providing significant reductions in search cost, comes at negligible space cost.

The remainder of this paper is organized as follows. Section 2 briefly reviews the literature on techniques that aim to speed up constraint satisfaction by exploiting learning. Section 3 presents the details of our approach. In Section 4 we present the experiments we carried out and analyse the results. In Section 5 a number of potential extensions of the work presented here are discussed. A number of concluding remarks are made in Section 6.

2 Background

One of the significant problems with constraint satisfaction techniques is the amount of search effort that may be required to solve a problem. Many techniques have been developed that try to minimize or avoid costly backtracking during search.

It has long been known that tree-structured constraint problems can be solved without backtracking [5]. This result has been extended to more general constraint networks also [6, 3]. However, these latter approaches are impractical due to their exponential worst-case space complexity.

Recent work in this area has combined automata theory and constraint satisfaction as a basis for supporting interactive configuration [1]. This approach essentially compiles a CSP into a form that can be used to answer particular questions without backtracking. Again, the disadvantage here is that the size of the automaton can grow exponentially depending on the properties of the constraint problem being considered.

Recent work on the notion of a general purpose backtrack-free representation for a CSP has been studied in the context of configuration [8]. This approach uses a fixed variable ordering to preprocess a CSP. By working in reverse through the variable ordering, values that could cause a backtrack are removed, so that backtracking never occurs when the CSP is being solved. The transformed CSP is called a backtrack-free representation (BFR). Of course, when values are removed, so are solutions. So in this approach the improvements in search efficiency come at the expense of lost solutions. While this is not acceptable in many domains, it is reasonable in some others where a solution needs to be found very quickly.

An alternative approach to improving the performance of search is the notion of nogood recording [18]. Nogoods are justifications of why particular instantiations of variables are inconsistent. By recording nogoods during search, backtracking can be minimized by exploiting this knowledge as search continues. Note that this is a form of learning.

In this paper an approach to improving the search for a solution is developed by relying on the inherent structure in the problem as a basis for learning how to characterize solutions. As we shall see, the proposed approach learns by inducing

a decision tree from examples and then relies on this as a basis for boosting search in the original CSP.

Decision tree learning is a standard machine learning technique for approximating discrete-valued functions [14, 13]. Decision trees make classifications by sorting the features of an instance through the tree from the root to some leaf node. Each leaf node provides the classification of the instance. At each node a test on some variable is performed, and each branch descending from that node corresponds to a possible outcome for that test. Upon reaching a leaf node a classification is made.

Decision tree techniques have been used by some researchers to reduce the length of dialog in interactive systems [4, 12]. However, note that here the purpose is different. We are focused on speeding up the process of answering a specific query in the context of system-initiative interaction.

The advantages of this approach are that we can regard the decision tree itself as a data structure, analogous to the principle underlying automaton-based approaches discussed above, but without the risk of incurring an exponential space cost, since we can bound the input to limit the size of the decision tree. Also, the decision tree can be seen as having some of the advantages of BFRs, specifically, an improvement in search cost, but without the penalty of solution loss. In the following section the details of our approach will be presented.

3 Boosting Constraint Satisfaction using Decision Trees

We present an approach to boosting the performance of search in an interactive context. We consider a limited form of interaction where a user constructs a query by making choices about some variables in the form of unary constraints. The task of the system is then to find a solution based on the user's query. If a solution is found it is presented to the user, otherwise he is notified that no solution exists.

The baseline in this work is a standard constraint solver, using inference and a good general-purpose heuristic. However, given that many real-world domains have structured solution spaces, we consider a combination of learning and constraint solving. Our approach involves inducing a decision tree from a set of solutions and non-solutions to the CSP we wish to interact with. Using this decision tree we extract a static variable ordering based on the occurrence of each variable in the tree. Users specify their queries based on this ordering of the variables.

3.1 Decision tree construction

In order to construct a decision tree, we need a number of exemplars of both 'classes' to be discriminated: solutions to the CSP (positive examples), and counterexamples – that is, fully instantiated tuples that are not solutions to the CSP.

Without making any strong assumptions about any model of the user, or wishing to too strongly reflect the structure of the problem, we wished to take

a sample of positive examples on an equiprobable basis from the totality of all solutions. For experimental purposes this was done by enumerating all solutions, and randomly selecting from those. Where in occasional cases the problem had less than the required number of solutions in total, the number of seeds was necessarily reduced. Part of the motivation for varying the number of seed solutions was to investigate the possible feasibility of a more sophisticated learning behavior being included in future development of this technique. Using this experimental variable allowed us to avoid the complications of assuming a particular user model, or of including incremental training of the tree without having first established an actual benefit.

Obtaining the negative examples is straightforward, as they represent for our chosen domain the considerable majority of the solution space, and can thus easily be generated randomly. The main remaining issue is how many of them to include in the training set; intuitively it seems reasonable to ‘train’ the system on more negative than positive examples, so as to get adequate discrimination of the latter in the tree. Equally, it is pragmatically necessary to bound the number of negative examples, as while they may have only a relatively small effect on the size of the final decision tree, a huge training set would adversely effect the performance of the tree construction algorithm. Somewhat arbitrarily, we have chosen throughout to use three times as many negative as positive examples in our set. Note that due to the nature of the tree construction algorithm, that works on sets of complete tuples and associated classifications, we cannot make direct use of ‘nogoods’ in the CSP sense, that is, partial tuples of which all possible extensions are inconsistent.

We construct decision trees using *ITI*, or *Incremental Tree Inducer* [19]. Contrary to the name, we use this software in a non-incremental or *direct metric* mode. Specifically, the tree is rebuilt in order to minimise a given criterion, in our case tree size. Tree size is critical in two senses: we are concerned that our (partial) representation of the CSP does not itself use an unacceptably large amount of space, and secondly the tree size will have an impact on the expense of decision-tree based search.

In the context of constructing the tree strictly in an off-line sense, and doing so only once for a whole training set, the considerable computational cost of doing this can be rationalized. However in other circumstances a trade-off may exist; in particular if instances become known gradually over time, and it is desirable to incorporate them in the decision tree, using the incremental facility may be more desirable for efficiency reasons. (The possibility still exists of using the direct metric restructuring periodically.)

3.2 Tree lookup

Given the above, we have constructed a binary tree where internal nodes represent (unary) comparisons on CSP variables, and the leaf nodes classify the variable instantiations corresponding to the path taken to reach it as (presumptively) satisfiable, or unsatisfiable. We can now ‘look up’ a partial tuple in the tree in the following manner: at each internal node, a variable is examined; if

that variable is assigned in the query tuple, we simply take the left or right branch according to whether the test on that value is verified or falsified. If the variable is *not* assigned in the query, we must consider *both* subtrees, accumulating different unary constraints in each case corresponding to the test in the node.

Thus in general we obtain a set of leaf nodes, each associated with a number of additional constraints, and a positive/negative judgement. We then solve in turn the sub-problem associated with each positive node, stopping as soon as we find one that is soluble (note that any assignments in the query not considered in the decision tree lookup phase must be satisfied in the subsequent search phase); if we find none, then we repeat the process on the *negative-judged* leaves similarly. This last is necessary to achieve accurate results as a presumptively negative region of the solution space may not in fact be solution-free, but simply lack a solution in the training set.

We account for the cost of lookups in terms of aggregated checks: each constraint check in each sub-problem is summed, together with the ‘decision tree checks’, that is simply the number of internal nodes traversed in reaching all relevant leaves.

3.3 Tree purification

In early experiments we observed that when the query was unsatisfiable, the results were consistently poor. This is due to the nature of the information encoded in the tree: while on the one hand, when we find a solution we can stop at once, if we do not, we must exhaust every leaf node in the sub-tree corresponding to the query, performing a search for each sub-problem. In general this is clearly less efficient than simply performing a single such search. In order to improve on this, it is evidently necessary to encode negative information in some way: to wit, which sub-problems are certain *not* to be soluble. In order to do this, we construct the tree as previously described, and then post-process it, as follows.

Having built a tree consistent with the given examples, we then analyze each negative node to determine if there is *any* extension of that node’s path leads to a solution in the problem at large. If there is not, we can label this node to so indicate: it is a ‘definite negative’ node. The consequences of this are that we can entirely eliminate such nodes from consideration when we encounter them while processing a query, and that if only one such node is found, we may immediately conclude. If the subproblem is in fact soluble, we leave the node’s annotation unchanged, and treat it subsequently exactly as we previously did all negative nodes. We do not add any solution found in this process to the tree, i.e. to the training-set used to build it, as this would require ‘splitting’ of an existing node, with furthermore no guarantee that the resultant new negative node would itself be ‘pure’.

Similar considerations do not arise with respect to the positive nodes; we make no attempt to verify that *all* of the tuples consistent with a given tree path correspond to solutions, as this would occur extremely rarely, and give us very

little query improvement, on anything other than massively underconstrained problems.

3.4 Query selection

A further consideration is the suitability of the query for the particular tree that happens to be constructed. In a second experiment with ‘purified’ trees, and arbitrary selection of query variables, results were still poor. On closer examination, it was clear that there was a considerable variance between queries that happened to correspond to the order variables were inspected in the tree, and those where the variables were ‘low down’ in the tree – or in extreme but not unlikely cases, not present at all in the tree. In such cases the number of nodes, and hence sub-problems, to be considered becomes large, and the performance consequently much worse, than if a ‘good’ choice of variable is made, and a small number of sub-problems is reached – ideally only one.

We considered two variations on a possible remedy for this; firstly, that we would use the tree *directly* to determine the order in which variables would be given values. (And thus, which variables *are* assigned a priori, for a query of a given length.) That is, given a value for the variables examined in the root node of the tree, then whichever variable we encounter consequent to that choice, etc.

Secondly, we considered a static approximation to the above. Assuming each value for each variable arises with equal probability, we calculate the *most likely* order for variables to be encountered in, given previous choices of instantiated variables. This is somewhat different conceptually in that it can be presented as a static variable ordering, with the choice of that variables need be instantiated done prior to any examination of the tree for a particular query.

Small-scale experiments revealed very little difference between these approaches, and a considerable improvement with both. For the somewhat larger-scaled experiments reported here, we chose to use only the ‘static’ order, that in principle is the weaker of the two in terms of improvement, but is a more natural approach in many query-based systems where a fixed set of questions is presented to a user.

In this instance, we began with the decision tree it was possible to construct with minimal size, and then used that to determine the nature of the queries considered, corresponding therefore to a ‘system initiative’ model. Equally however, had we prior knowledge of which queries were to be considered, or more likely to be considered, we could have used that information to guide construction of the tree.

4 Experiments and Evaluation

4.1 Setup

Problem sets. Given that our motivation lies in the realm of configuration and other similar problems, we sought to generate problems that were in the

first instance soluble, and secondly had a somewhat large number of solutions, ideally without being trivial to solve.

We consequently chose to look at problems with number of solutions expected to be in the region of thousands or tens of thousands, and having generated these, excluded those that were insoluble, and those that had many more solutions (greater than 100,000).

For our application domain, problems will typically be relatively sparse (have a low proportion of the maximum number of constraints possible) and less tight (allow many consistent assignments of values to variables). However, we did not wish to restrict our attention to such problems, and thus included a range of densities, each coupled with an associated tightness value to give ‘solubility’ values, as per Table 1. All problems had 30 variables with 10 domain values for each.

Table 1. Problems used in our experiments.

density (number of constraints)	37	50	65	86	117	167	268
tightness	0.8	0.7	0.6	0.5	0.4	0.3	0.2

The problem generator. We used the problem generator of Zou et al [21, 2]. In addition to the conventional parameters of problem size, tightness, and density, an additional value is given to control the degree of interchangeability [7] of the problem, in the following sense: the *induced degree of fragmentation* (IDF) of a constraint measures the number of equivalence classes a binary constraint has with respect to *one* of its variables. This is thus a form of *NIC*, Neighbourhood Interchangeability per constraint [2, 10]. The Zou generator yields a problem in which each constraint has the specified IDF.

Reference solver. In order to conduct our experiments, a baseline solver is required. Similarly, having identified sub-problems from the decision tree, we must have the facility to solve those. To some extent therefore, we can remove some possible bias from any comparison by using the same solver in each case, thus getting a relatively ‘like with like’ comparison modulo the cost of the decision tree itself that was anticipated (and proved) to be relatively low. (We equate decision tree tests and constraint checks one for one, conservatively, since the former are the more atomic of the two.)

Two of the most common algorithms used to solve CSPs are forward-checking (FC) [9] and maintain arc-consistency (MAC) [16]. We chose to use a solver with FC, on the basis of some early experiments where we found that FC outperforms MAC on problems with numbers of solutions in the range we were concerned with – that is, some way away from the peak of difficulty. We used minimum domain over forward degree as a variable ordering heuristic, and lexicographically chose values from within each domain.

Experimental parameters. We conducted experiments over a range of problems, with 30 variables, domain size of 10, and with the range of tightness and density noted earlier in this section. We chose the practicable extrema for values of IDF: the maximum is the domain size, 10, and the minimum we found in practice that could be reliably generated over this range of parameters, of IDF=3 (corresponding to the greatest degree of interchangeability). For each parametric point, 20 problems were generated with the required characteristics. We conducted each experiment with 10, 100, and 1000 positive seeds, in turn. As already noted, we have chosen throughout to use three times as many negative seeds as positive ones. Query sizes of 1, 2, 3, 5 and 7 were used successively, and for each of these 1000 such queries were generated.

4.2 Discussion of the Results

The results of our experiments are presented in Figure 1. We present a graph for each query length. The x-axis in each case presents the number of constraints in the problem, the y-axis represents the saving in the number of constraint checks required to find a solution satisfying the user’s query or proving that none exists. The improvement per query is measured as the difference between the totals checks required in each case, divided by the greater of the two numbers to give a properly scaled fractional improvement measure:

$$improvement = \frac{cc_s - cc_t}{max(cc_s, cc_t)}$$

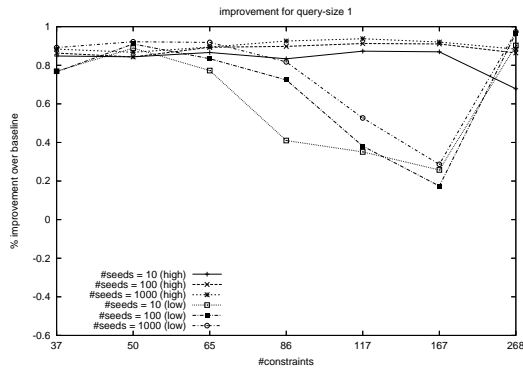
where cc_s is the number of constraint checks required by the constraint solver alone, and cc_t is the number of constraint checks required by the hybrid decision tree and constraint solver approach.

From this we take the arithmetic mean over every query for each problem to obtain the improvement per problem. We plot the median of the means over all problems.

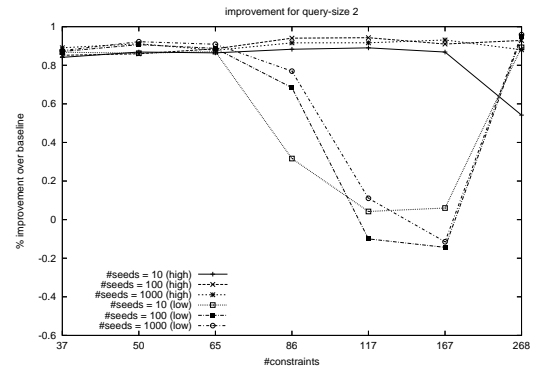
It is clear from these graphs that for high interchangeability (that is, numerically low values of IDF) our improvement is relatively consistent, and high throughout, essentially independently of both remaining problem parameters and query size. We consider this significant, as in many senses such problems are more characteristic of ‘real’ problems than are purely random ones. Even for low interchangeability problems (i.e., high IDF) we still see consistently high improvements for the lower half of the density/tightness scale, and for the very highest such point.

The number of solutions used to build the tree has little impact throughout. This has implications for a more explicitly learning-based approach, in that it seems to indicate we may be able to get the bulk of the possible improvement for relatively little initial cost in learning, at least where the ‘training examples’ have in some sense good coverage of the whole problem space (as here tends to be the case due to random selection).

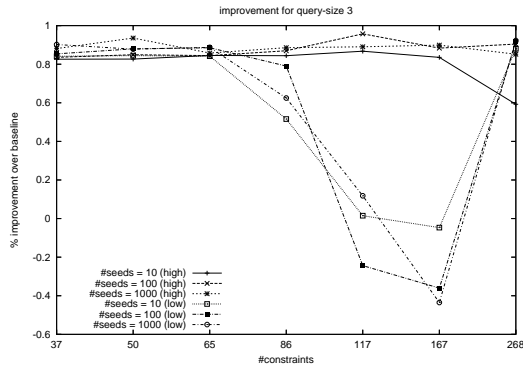
The only consistent feature of the effect of the number of examples, is that we can see a consistent deterioration in performance for the largest number of



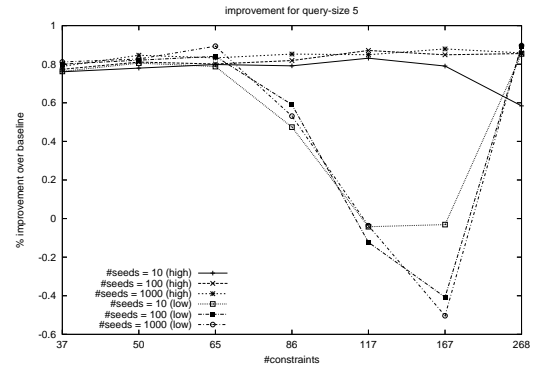
(a) Query size = 1



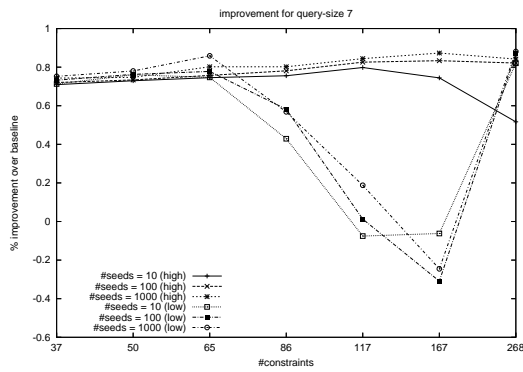
(b) Query size = 2



(c) Query size = 3



(d) Query size = 5



(e) Query size = 7

Fig. 1. Median fractional improvement in constraint checks by using the decision tree approach against the baseline constraint solver (forward checking using min-domain/forward degree as a variable ordering heuristic).

constraints in each graph in Figure 1. However, once the number of examples used to build the tree is increased, this effect is no longer present.

One of the most dramatic features of our results is the size of the decision tree that we acquire in these problems. Figure 2 presents the median tree size, expressed as the number of internal nodes in the tree, for each problem-set. Clearly, problems with high interchangeability have consistently small decision trees. Recall that these trees give us an almost consistent order-of-magnitude improvement in search cost. On the other hand, problems with low interchangeability require larger trees, and increasing the number of solutions used to build such trees makes matters worse, particularly for problems with larger numbers of constraints. This is to be expected, of course, since when the solutions to a problem exhibit high amounts of interchangeability, fewer tests on variables are required to correctly classify the training data. Therefore, in the context of real-world problems, that often have high interchangeability amongst their solutions, the approach presented here gives a very small data structure that provides significant savings in search effort.

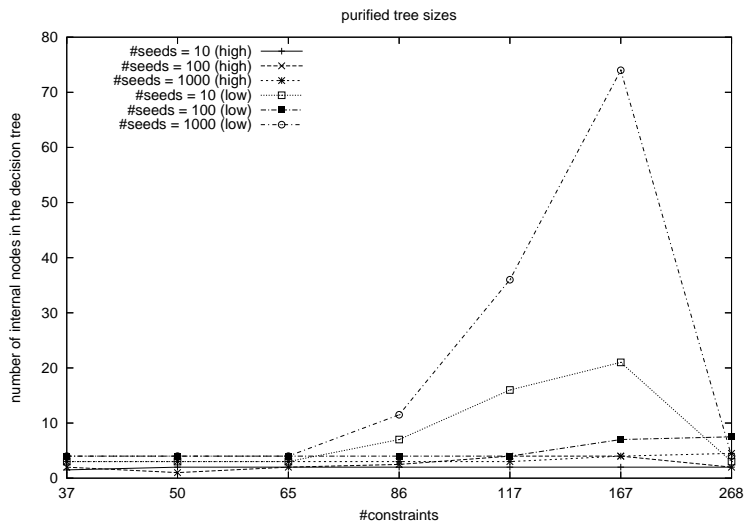


Fig. 2. The median sizes of the decision tree used for each problem-set.

In terms of the motivating scenarios outlined at the outset, these results are extremely positive. At the cost of finding a number of solutions to a large problem, and using a restricted ordering of the variables when constructing queries, significant reduction in search effort can be obtained. In an interactive context these results are very significant since response time can be reduced by almost an order-of-magnitude. Speed-of-response is the critical issue when dealing with any form of interactive system.

5 Extensions

In this paper we have focused on introducing a new approach to solving constraint satisfaction problems in a restricted interactive context. The primary objective here was to show that finding a solution using the hybrid approach of using a decision tree as a first-phase of search was fruitful. Having established that result, there are a number of directions in which we can take this work, some of which we discuss briefly here.

By varying the number of seed solutions used to build the decision tree, we have seen that there is evidence that supports the hypothesis that learning over time (using more solutions) can improve performance. However, as we saw, only marginally so after some point: the utility we received by increasing the number of seed solutions from 10 to 100 was more than for 100 to 1000. Therefore, all this shows at the moment is that we can benefit quite a lot from a small amount of learning. In the future, it would be interesting to study the particular choice of seed solutions to be used when building the decision tree. In particular, if we initially constructed the decision tree on a random basis with a small number of solutions, but could characterize good solutions to queries that should be used to modify the tree, we may find that we can get even better improvements in search with even smaller trees.

Furthermore, we have discussed how our results improved when we could identify pure negative leaves in the decision tree when processing a query. In a similar fashion we can associate an example solution with each positive leaf node in the tree. This will help improve the speed with which we can answer queries in which all user-specified constraints have been considered upon reaching a positive leaf. We have not studied this enhancement in the experiments we have presented here since we did not want to complicate the discussion with considerations related to the storage of solutions.

Another important direction, is to use the purified decision tree that we use here as a classifier for user queries. Currently, we use both the decision tree and a constraint solver to find a solution to a user's query. Therefore, we are always guaranteed to find the correct answer: either a solution exists that satisfies the query and we find it, or none exists and we discover that. This completeness comes from the fact that we resort to search when there is ambiguity. Another direction that we could consider, of course, is to only rely on the decision tree for solving the decision problem. Specifically, in many applications a user just wants to know if a solution exists that satisfies a query or not. We could simply rely on the decision tree for this task, if we are prepared to accept some number of incorrect classifications. For example, for a given query, if the decision tree has a positive leaf node (classifying a query as having a solution) having considered every unary constraint in the query, then this is sufficient to report that there is at least one solution to this query and that the decision problem has a positive outcome (a solution exists). On the other hand, if we reach a purified negative leaf node, then the decision problem is solved in the negative (a solution does not exist). Reaching an 'impure' negative node we cannot be certain whether or

not a solution exists, but we have some confidence in committing to a particular decision.

This latter extension can be very useful in many real-world applications where only the existence or non-existence of a solution is required. Given the sizes of decision trees we have observed above, this is a facility that could be provided with very low overhead.

6 Conclusions

In this paper we have presented an approach that uses known solutions to a problem to improve search. The approach we present is based on a combination of decision tree learning and constraint satisfaction. We demonstrated that close to order-of-magnitude improvements in search effort can be achieved using this hybrid approach over traditional search. Furthermore we have shown that the space complexity using this approach is almost negligible.

The approach is limited to scenarios where the system takes the initiative by asking a user to provide choices for a particular subset of the variables in the problem. We are currently studying ways of relaxing this restriction.

Based on the positive preliminary results we have obtained here, we will be studying our approach in the context of real world configuration problems, as well as in other interactive problem-solving contexts.

References

1. J. Amilhastre, H. Fargier, and P. Marguis. Consistency restoration and explanations in dynamic cps – application to configuration. *Artificial Intelligence*, 135:199–234, 2002.
2. A.M. Beckwith, B.Y. Choueiry, and H. Zou. How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In *AI 2001: Advances in Artificial Intelligence, 14th Australian Joint Conference on Artificial Intelligence. Lecture Notes in Artificial Intelligence LNAI 2256*, pages 50–61, Adelaide, Australia, 2001. Springer Verlag.
3. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.
4. M. Doyle and P. Cunningham. A dynamic approach to reducing dialog in on-line decision guides. In *Proceedings of EWCBR*, pages 49–60, 2000.
5. E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
6. E.C. Freuder. Complexity of k-tree-structured constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 4–9, 1990.
7. E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the AAAI*, pages 227–233, 1991.
8. E.C. Freuder, T. Carchrae, and J.C. Beck. Satisfaction guaranteed. In *Proceedings of the IJCAI-2003 Workshop on Configuration*, 2003.
9. R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(4):263–313, 1980.

10. A. Haselböck. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of the 13th IJCAI*, pages 282–287, 1993.
11. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
12. D. McSherry. Minimizing dialog length in interactive case-based reasoning. In *Proceedings of the 17th IJCAI*, pages 993–998, 2001.
13. T. Mitchell. Decision tree learning. In *Machine Learning*, chapter 3, pages 52–80. McGraw Hill, 1997.
14. J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.
15. Dechter. R. *Constraint Processing*. Morgan Kaufmann, 2003.
16. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 125–129, 1994.
17. D. Sabin and R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, July–August 1998. Special Issue on Configuration.
18. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(2):187–207, 1994.
19. P.E. Utgoff, N.C. Berkman, and J.A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5–44, 1997.
20. M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1–2):139–168, 1996.
21. H. Zou, A.M. Beckwith, and B.Y. Choueiry. A Generator of Random Instances of Binary Finite Constraint Satisfaction Problems with Controllable Levels of Interchangeability. Technical Report CONSYSTLAB-01-01. Available from <http://consystlab.unl.edu/CSL-01-01.doc>, University of Nebraska-Lincoln, 2001.

An Empirical Study of a New Restart Strategy for Randomized Backtrack Search

Venkata Praveen Guddeti and Berthe Y. Choueiry

Constraint Systems Laboratory
Computer Science & Engineering
University of Nebraska-Lincoln
Email: {vguddeti, choueiry}@cse.unl.edu

Abstract. We propose an improved restart strategy for randomized backtrack search and compare its performance to other search mechanisms in the context of solving a tight real-world resource allocation problem. The restart strategy proposed by Gomes et al. [1] requires the specification of a cutoff value determined from an overall profile of the cost of search for solving the problem. When no such profile is known, the cutoff value is chosen by trial-and-error. Walsh proposed the strategy Randomization and Geometric Restart (RGR), which does not rely on a cost profile but determines the cutoff value as a function of a constant parameter and the number of variables in the problem [2]. Unlike these strategies, which have fixed restart schedules, our technique (RDGR) dynamically adapts the value of the cutoff parameter to the results of the search process. We empirically evaluate the performance of RDGR by comparing it against a number of heuristic and stochastic search techniques, including RGR, using the cumulative distribution of the solutions. We compare the performance of RGR and RDGR over different run-time durations, different values of the cutoff, and for different problem types (i.e., a real-world resource allocation problem and randomly-generated binary constraint satisfaction problems). We show that distinguishing between solvable and over-constrained problem instances in our real-world case-study yields new insights on the relative performance of the search techniques tested. We propose to use this characterization as a basis for building new strategies of cooperative, hybrid search.

1 Introduction

We apply Constraint Processing techniques to model and solve the assignment of Graduate Teaching Assistants (GTA) to courses in our department. The idea of this application was first found on the web-page of Rina Dechter. This is a critical and arduous task that the department's administration has to drudge through every semester. By focusing our investigations on this particular real-world application, we have been able to identify and compare the advantages and shortcomings of the various search strategies we have implemented to solve this problem. Such an insight is unlikely to be gained from testing toy problems, and surely difficult from testing random problems. We believe that the identified behaviors apply beyond our application and are currently working on testing this hypothesis. The contributions of this paper are as follows: (1) The development

of a new dynamic restart strategy for randomized backtrack search, and (2) an empirical evaluation of the performance of this new strategy and a comparison with other heuristic and stochastic search techniques on a real-world problem and on randomly generated binary CSPs.

This paper is structured as follows. Section 2 describes the GTA assignment problem and our implementations of a backtrack search, a local search, and a multi-agent search technique for solving it. Section 3 introduces our new proposed dynamic restart strategy for randomized backtrack search and our implementation of Walsh’s restart strategy [2]. Section 4 presents our experiments and our observations. Finally, Section 5 concludes the paper and provides directions for future research.

2 GTA Assignment Problem

Given a set of graduate teaching assistants (GTAs), a set of courses, and a set of constraints that specify allowable assignments of GTAs to courses, the goal is to find a consistent and satisfactory assignment [3–6]. Hard constraints (e.g., a GTA’s competence, availability, and employment capacity) must be met, and GTA’s preferences for courses (expressed on a scale from 0 to 5) must be maximized. Typically, every semester, the department has about 70 different academic tasks and can hire between 25 and 40 GTAs. Instances of this problem, collected since Spring 2001, are consistently tight and often over-constrained. However, *this is not known a priori*. The objective is to ensure GTA support to as many courses as possible by finding a *maximal consistent partial-assignment*. Because the hard constraints cannot be violated, the problem cannot be modeled as a MAX-CSP [7]. We provide a constraint model of this problem by representing the courses as variables, the GTAs as domain values, and the assignment rules as a number of unary, binary, and non-binary constraints. We define the problem as the task of finding the longest assignment, as a primary criterion, and maximizing GTAs’ preferences, as a secondary criterion. (We model the latter as the value of the geometric mean of GTAs’ preferences in an assignment.) We implemented a number of search strategies for solving this problem, which we summarize below. These are a heuristic backtrack search (BT) with various ordering heuristics, a greedy local search (LS), a multi-agent-based search (ERA), and a randomized backtrack search with two restart strategies (RGR and RDGR). All strategies implement the above two optimization criteria, except ERA, which models the GTA assignment problem as a satisfaction problem. We tested these search techniques on the real-world data-sets shown in Table 1. Each course has a load that indicates the weight of the course. For example, a value of 0.5 means this course needs one-half of a GTA. The *total load* of a semester is the cumulative load of the individual courses. Each GTA has a capacity factor which indicates the maximum course weight he/she can be assigned during the semester. The sum of the capacities of all GTAs represents the *total capacity*.

Below, we review the search techniques to which we compare our new dynamic restart strategy. These search techniques were implemented separately by students, competing to produce the best results.

Data set	Spring2001b	Fall2001b	Fall2002	Fall2002-NP	Spring2003	Spring2003-NP
Reference	1	2	3	4	5	6
Solvable?	×	√	×	×	√	√
#Courses (#variables)	69	65	31	59	54	64
#GTAs (domain size)	26	34	28	28	34	34
Total capacity	26	30	11.5	27	27.5	31
Total load	29.6	29.3	13	29.5	27.4	30.2
Ratio = $\frac{\text{TotalCapacity}}{\text{TotalLoad}}$	0.88	1.02	0.88	0.91	1.00	1.02

Table 1. Characteristics of the data sets.

2.1 Heuristic backtrack search

Our heuristic backtrack (BT) search is a depth-first search with forward checking [8]. Because the problem may be over-constrained, we modified the backtrack mechanism to allow null assignments and proceed toward the longest solution in a branch-and-bound manner (i.e., backtracking is not performed when a domain is wiped-out as long as there are future variables with no empty domains). Our implementation is described in detail in [4]. Note that adding dummy values to deal with over-constrained instances is a bad choice in our context as it increases the branching factor (which is already too large) and consequently the worsens the thrashing behavior.

We have also implemented several variable and value ordering heuristics to improve the performance of search. For variable ordering we implemented two heuristics for choosing the most constrained variable first: least domain and ratio domain size to degree. We applied these heuristics both statically (i.e., sequence of variables is determined before search and not modified thereafter) and dynamically (i.e., the next variable is chosen after each instantiation). For value ordering, we tested 3 different heuristics: random ordering, and sorted by preference and by occurrence frequency in the domains. The combination of these heuristics yielded 12 ordering strategies. Our experiments showed that dynamic variable ordering is consistently superior to static ordering, but that the influence of the other factors is not significant in the context of our application.

Furthermore, all these strategies exhibited a serious vulnerability to thrashing, which seriously undermined their ability to explore wider areas of the search space. Indeed, although BT is theoretically sound and complete, *the size of the search space makes such guarantees meaningless in practice*. Figure 1 illustrates thrashing for a problem with 69 variables and 26 values. Here, the percentage is $\frac{\text{number of variables} - \text{shallowest level}}{\text{number of variables}}$. Indeed, the shallowest level of backtrack achieved after 24 hours (26%) is not significantly better than that reached after 1 minute (20%) of search, never revising the initial assignment of 74% of the variables. Figure 2 shows, for each data set, the number of variables, the longest solution (max depth), and the shallowest BT levels in terms of the level and the percentage of backtracking in the search tree attained after 5 minutes and 6 hours.

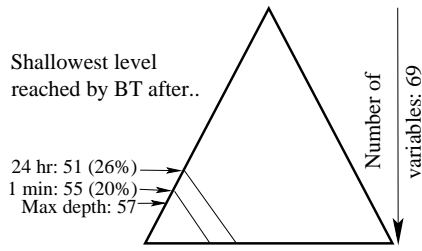


Fig. 1. BT search thrashing in large search spaces.

Data set	# Vars	BT running for..			
		5 min		6 hours	
		Max depth	Shallowest level %	Max depth	Shallowest level %
1	69	57	53 23%	57	51 26 %
2	65	63	55 15%	63	54 16 %
3	31	28	13 58%	28	3 90 %
4	59	49	48 18%	50	45 23 %
5	54	52	44 18%	54	41 24 %
6	64	62	54 15%	62	47 26 %

Fig. 2. BT search thrashing.

2.2 Local search

Zou and Choueiry designed and implemented a greedy, local search (LS) technique to solve the GTA Assignment Problem [9–11]. It is a hill-climbing search using the min-conflict heuristic for value selection [12]. It begins with a complete, random assignment (not necessarily consistent) and tries to improve it by changing inconsistent assignments in order to reduce the number of constraint violations. The effects of consistent assignments are propagated over the domains of the variables with inconsistent assignments. This design decision effectively handles non-binary constraints. Also, the local search is greedy in the sense that consistent assignments are not undone. Moreover, a random-walk strategy is applied to escape from local optima [13]. With a probability $(1 - p)$, the value of a variable is chosen using the min-conflict heuristic, and with probability p this value is chosen randomly. Following the indications of [13], $p = 0.02$ is used. Finally, random restarts are used to break out of local optima.

2.3 ERA model

Zou and Choueiry also implemented a multi-agent-based search for solving the GTA Assignment Problem [9–11]. Liu et al. [14] proposed the ERA algorithm (Environment, Reactive rules, and Agents), a multi-agent-based search for solving CSPs. Each agent represents a variable. The positions of an agent in the environment E correspond to the values in the domain of the variable. First, ERA places the agents randomly in their allowed positions in the environment, then it considers each agent in sequence. For a given agent, it computes the constraint violations of each agent’s position. An agent moves to occupy a position (zero position) that does not break any of the constraints that apply to it. If the agent is already in a zero position, no change is made. Otherwise, the agent chooses a position to move to, the choice being determined stochastically by the reactive rules (R). The agents keep moving until they all reach a zero position (i.e., a full, consistent solution) or a certain time period has elapsed. After the last iteration, only the CSP variable corresponding to agents in zero position are effectively instantiated. The remaining ones remain unassigned (i.e., unbounded). This algorithm acts as an ‘extremely’ decentralized local search, where any agent can move to any position, likely forcing other agents to seek other positions.

This extreme mobility of agents in the environment is the reason for ERA's unique immunity to local optima, as uncovered by the experiments in [9–11]. It is indeed the only search technique to solve instances that remain unsolved by any other technique we tested. Zou and Choueiry also uncovered the weakness of ERA on over-constrained problems, where a deadlock phenomenon undermines its stability resulting in particularly short solutions. However, it has been shown that this phenomenon can be advantageously used to isolate, identify, and represent conflicts in a compact manner.

3 Randomized BT search with restarts

Unlike ERA and local search, general backtrack (BT) search has the nice property of being complete and sound. However, the performance of heuristic BT proved to be unpredictable in practice and seriously undermined by thrashing (i.e., searching unpromising parts of the search space). Thrashing can be explained by incorrect heuristic choices made early in the search process, and forces BT search to explore large 'barren' parts of the search tree. As the problem size increases, the effects of thrashing become more important, and the performance of search dramatically decreases. Another major problem is the high degree of unpredictability in the run-time of BT over a set of problem instances, even within the same problem type. Gomes et al. [1] noticed that this run-time can be often modeled by a heavy-tailed distribution. They proposed to use randomization and restart strategies to overcome this shortcoming of systematic search. First we review the main concepts, then we describe the two strategies that we tested.

Gomes et al. [1] demonstrated that randomization of heuristic choices combined with restart mechanisms is effective in overcoming the effects of thrashing and in reducing the total execution time of systematic BT search. Thrashing in BT search indicates that search is stuck exploring an unpromising part of the search space, and thus incapable of improving the quality of the current solution. It becomes apparent that there is a need to interrupt search and to explore other areas of the search space. It is important to restart search from a different portion of the search space; otherwise it will end up traversing the same paths. Randomization of branching during search is used to this end. Randomness can be introduced in the variable and/or value ordering heuristics, either for tie-breaking or for variable and/or value selection. After choosing a randomization method, the algorithm designer must decide on the type of restart mechanism. This restart mechanism determines when to abandon a particular run and restart the search. Here the tradeoff is that reducing the cutoff time reduces the probability of reaching a solution at a particular run. Several restart strategies have been proposed with different cutoff schedules. Some of the better known ones are the fixed-cutoff strategy and Luby et al.'s universal strategy [15], the randomization and rapid restart (RRR) of Gomes et al. [1], and the randomization and geometric restarts (RGR) of Walsh [2]. Among the above listed restart strategies, RRR and RGR have been studied and empirically tested in the context of CSPs. All of these restart strategies are static in nature, i.e. the cutoff value for each restart is independent of the progress made during search. Some restart strategies (e.g., fixed-cutoff strategy of [15] and RRR [1]) employ an optimal cutoff value that is fixed for all the restarts of a particular problem instance.

However, the estimation of the optimal cutoff value requires a priori knowledge of the cost distribution of that problem instance, which is not known in most setting and must be determined by trial-and-error. This is clearly not practical for real-world applications. There are other restart strategies that do not need any a priori knowledge (e.g., Luby et al.’s universal strategy [15] and Walsh’s RGR [2]). They utilize the idea of an increasing cutoff value in order to ensure the completeness of the restart strategy. However, if these restart strategies do not find a solution in the initial few restarts, then the increasing cutoff value leads to fewer restarts, which may yield thrashing and diminishes the benefits of the restart strategy. We propose a restart strategy that dynamically adapts the cutoff value for each restart based on the performance of previous restarts. We do this at the expense of completeness. We also implemented RGR and empirically compared it with our dynamic restart strategy.

3.1 Randomization and Geometric Restarts

Walsh proposed the Randomization and Geometric Restarts (RGR) strategy to automate the choice of the cutoff value [2]. According to RGR, search proceeds until it reaches a cutoff value for the number of nodes visited. The cutoff value for each restart is a constant factor, r , larger than the previous run. The initial cutoff is equal to the number of variables n . This fixes the cutoff value of the i^{th} restart at $n \cdot r^i$ nodes. The geometrically increasing cutoff value ensures completeness with the hope of solving the problem before the cutoff value increases to a large value. We studied various values of r and report them in this paper. We combined this restart strategy with the backtrack search of Section 2.1, randomizing the selection of variable-value pairs.

3.2 Randomization and Dynamic Geometric Restarts

We now introduce a simple but effective improvement to RGR. Static restart strategies suffer from the problem of increasing cutoff values after each restart. While this ensures completeness of the search, it results in fewer restarts, thus increasing the likelihood of thrashing and diminishing the probability of finding a solution. Our proposed strategy, Randomization and *Dynamic* Geometric Restarts (RDGR), aims to attenuate this effect. It operates by not increasing the cutoff value for the following restart whenever the quality of the current best solution is not improved upon. When the current restart improves on the current best solution, then the cutoff value is increased geometrically, similar to RGR. Because the cutoff value does not necessarily increase, completeness is no longer guaranteed. This situation is acceptable in application domains (like ours) with large problem size where completeness is, anyway, infeasible in practice. Smaller cutoff values result in a larger number of restarts taking place in RDGR than RGR, which increases the probability of finding a solution. All other implementation details are similar to RGR.

Let C_i be cutoff value for the i^{th} restart and r be the ratio used to increase the cutoff value. In RGR the cutoff value is updated according to the equation: $C_{i+1} = r \cdot C_i$. We use the following equation in RDGR:

$$C_{i+1} = \begin{cases} r \cdot C_i & \text{when the solution has improved at the } i^{th} \text{ restart} \\ C_i & \text{otherwise} \end{cases} \quad (1)$$

In RGR, the cutoff value for each restart is determined *independently* of how search performed at the previous step. However, this is not the case for RDGR. Each time search begins with a different random seed, it traverses different search paths. Some paths may be more fruitful than others. RGR and RDGR follow the same cutoff schedules for search paths that improve solutions. When this is not the case, RGR cutoff values keep on increasing, thus making RGR more of a randomized BT search than a randomized BT search with restarts. In contrast, RDGR keeps cutoff at smaller values. This explains the dynamic nature of RDGR. For problems that are not tight, solutions are found within a few restarts. In such cases, RGR and RDGR exhibit similar behaviors. For tight and over-constrained problems, RDGR seems to dominate RGR as we show in our experiments (Section 4).

4 Experiments and results

We tested and compared the above listed 5 search strategies, namely: BT (Section 2.1), LS (Section 2.2), ERA (Section 2.3), RGR (Section 3.1), and RDGR (Section 3.2). BT is deterministic and the other 4 search techniques (i.e., LS, ERA, RGR, and RDGR) are stochastic. In the terminology introduced by Hoos and Stützle in [16], these are optimization Las Vegas algorithms, RGR is probabilistically approximately complete (PAC), and LS, ERA, and RDGR are essentially incomplete. We tested these search techniques on the 6 real-world data-sets of the GTA Assignment Problem shown in Table 1. Three of the data sets (1, 3, and 4) are over-constrained, and the remaining ones (2, 5, and 6) are tight but solvable. Table 2 shows the performance of BT on data set 1 for various run times.

Data set 1 (69 variables, over-constrained)						
CPU run time	30 sec	5 min	30 min	1 hour	6 hours	24 hours
Shallowest BT level	54	53	52	52	51	51
Longest solution	57	57	57	57	57	57
Geometric mean of preference values	2.15	2.17	2.17	2.21	2.27	2.27
# Backtracks	1835	47951	261536	532787	3274767	13070031
# Nodes visited	3526	89788	486462	989136	6059638	24146133
# Constraint checks	84961180	316839736	1813726855	3584407690	21569911975	86996547613

Table 2. Performance of BT for various CPU run-times.

We repeated our experiments 500 times for all stochastic search procedures. Naturally, a single run is sufficient for BT because it is deterministic. We found that the average run-time for all stochastic algorithms stabilizes after 300 runs on all our data, as shown in Figure 3 for data set 1, which justifies our decision. We experimented with different run-times for each run of each algorithm. We also experimented with different ratios used to increase the cutoff value in RGR and RDGR. We compare the performance of the algorithms using the following criteria:

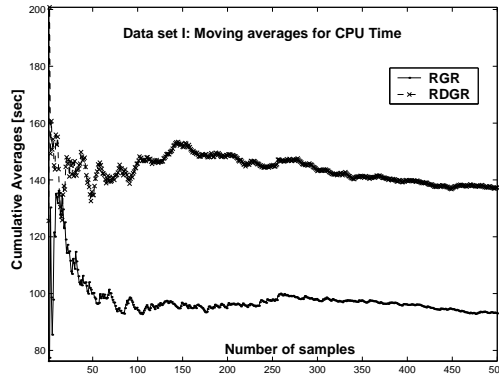


Fig. 3. Moving average for CPU run-times for data set 1.

1. *Solution quality distributions (SQD)* taking as reference the longest known solution for each data set, as recommended by Hoos and Stützle in [16]. SQD's are cumulative distributions of the solution quality, similar to the cumulative distributions of run-time in run-time distributions. The horizontal axes represents in percent the relative deviation of the solution size s from the longest known solution s_{opt} , computed as $\frac{(s_{opt}-s)100}{s_{opt}}$. Thus, the point 0% on the x -axis denotes the longest solution and, the point 20% denotes a solution that is 20% shorter than the longest solution.
2. *Descriptive statistics* of all the solutions found, for all search techniques.
3. *95% confidence interval* of the mean improvement using 25 mean sample points, each sub-sample being of size 20. The confidence interval was computed using a t -distribution. The improvements of RDGR with respect to an algorithm A are computed as:

$$\text{Improvement}(X) = \frac{X(A) - X(\text{RDGR})}{X(A)} \quad (2)$$

where X is deviation from the best known solution in percentage. Table 3 reports the improvements of RDGR over RGR and ERA.

We report the results for the following data sets (the same qualitative observations hold across all data sets):

- Data set 1 as a representative of an over-constrained problem. Results are shown in Figures 4, 5, 6, and 7, and Table 4.
- Data set 5 as a representative of a tight but solvable problem. Results are shown in Figures 8, 9, 10, and 11, and Table 5.

We also evaluated all the search techniques on randomly generated problems, generated with the model B type generator of [17]. We generated three types of randomly generated problems, each containing 100 instances and each instance run for 3 minutes:

- The first type of randomly generated problems (R1) are *under-constrained* binary CSPs with 40 variables, uniform domain size of 20 values, 0.5 constraint proba-

Data set	Improvements over RGR			Improvements over ERA		
	LL	Average	UL	LL	Average	UL
1	1.83	2.23	2.63	45.47	46.26	47.06
2	1.19	1.48	1.78	-5.64	-5.17	-4.69
3	2.61	2.94	3.27	30.05	32.37	43.69
4	1.03	1.35	1.66	24.71	26.70	28.70
5	0.61	0.84	1.08	-3.54	-3.38	-3.23
6	0.86	1.15	1.45	-2.47	-1.91	-1.36

LL: lower limit of the confidence interval.
UL: upper limit of the confidence interval.

Table 3. Improvements of RDGR with 95% confidence level

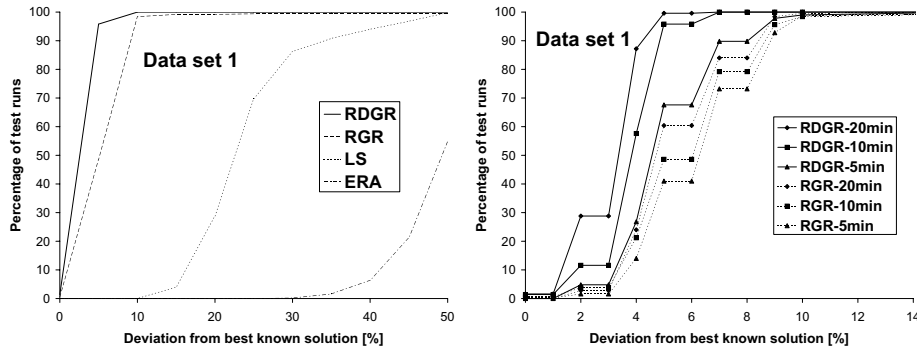


Fig. 4. SQD for data set 1 (500 runs, 10 min each). **Fig. 5.** RGR and RDGR over different run-times for data set 1 (500 runs).

bility, and 0.2 constraint tightness. We give their results in Figures 12, 13 and 14.

- The second type of randomly generated problems (R2) are *over-constrained* binary CSPs with 40 variables, uniform domain size of 20 values, 0.5 constraint probability, and 0.5 constraint tightness. We give their results in Figures 15, 16, 17, and 18.
- The third type of randomly generated problems are from the *phase transition* area. These are binary CSPs with 25 variables, uniform domain size of 15 values, 0.5 constraint probability, and 0.36 constraint tightness. They were split into two sets, each of 100 instances. The first set (R3) are solvable, while the second set (R4) are not solvable. We give their results in Figures 19, 20, 21 (RDGR), 22 (RDGR), 23 (RGR), and 24 (RGR).

Below we report our observations:

Improvement of RDGR over RGR: Figures 4, 5, 8, 9, 12, 15, 19, and 20 show that RDGR clearly improves upon RGR. In Figures 4 and 8, RDGR has greater probability of finding solutions up to 10% relative solution size. After that value, RDGR

Data set 1 (69 variables, over-constrained)						
Search	Mean	Median	Mode	Standard dev.	Minimum	Maximum
BT	57	57	57	0	57	57
LS	47.12	48	49	4.44	30	55
ERA	30.99	31	32	4.37	18	45
RDGR	59.66	60	60	0.77	58	62
RGR	58.27	58	58	2.83	23	62

Table 4. Statistics of solution size (500 runs, 10 min each).

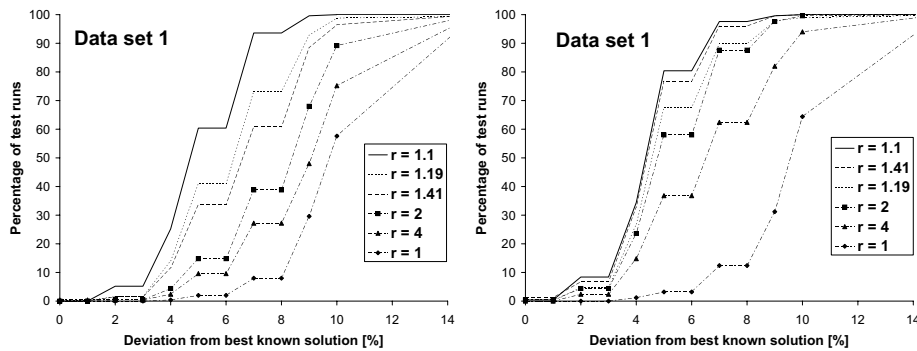


Fig. 6. SQRs of RGR with different ratios for cut-off value (500 runs, 5 min each). **Fig. 7.** SQRs of RDGR with different ratios for cutoff value (500 runs, 5 min each).

and RGR have similar performances. In Figures 5 and 9, RDGR consistently performs better than RGR over different run-times. The positive improvements in Table 3 show that RDGR performs better than RGR over all the GTA data sets.

Best results on the ratio used to increase the cutoff: In accordance with [2], Figures 6, 10, 13, 17, 23, and 24 show that a value of $r=1.1$ is the best among the values tested for RGR. While, for RGR, this optimal ratio does not change with the problem type (i.e., GTA vs. random problem), it does for RDGR. For the GTA problem, it is $r=1.1$ (Figures 7 and 11). For randomly generated problems, it is $r=2$ (Figures 14, 18, 21, and 22).

Improvement of RDGR over BT: Tables 4 and 5 show that the maximum value of the solution sizes produced by RDGR is clearly greater than that of the solution sizes produced by BT. However, due to its stochastic nature, RDGR suffers from high instability in its solution quality. On randomly generated problems also, RDGR outperforms BT (Figures 12, 15, 19, and 20).

Superiority of RDGR over LS: The performance of RDGR is clearly superior to that of LS (see Tables 4 and 5, and Figures 4, 8, 12, 16, 19, and 20). Although this solution quality is highly variable for both RDGR and LS, the low mean value of the solution quality of LS ensures that RDGR remains superior to LS.

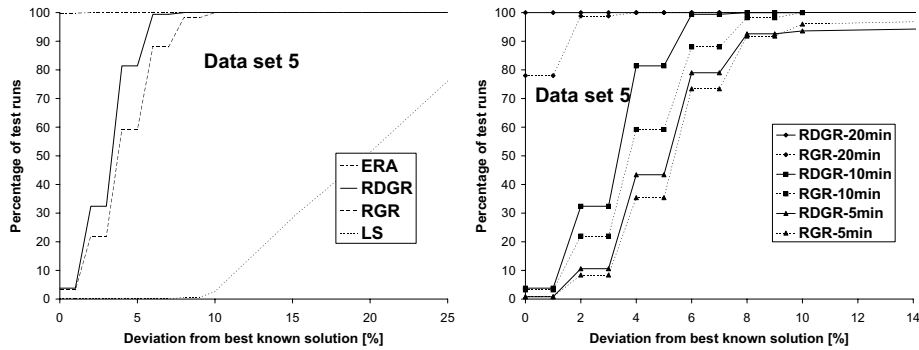


Fig. 8. SQR for data set 5 (500 runs, 10 min each). **Fig. 9.** RGR and RDGR over different run-times for data set 5 (500 runs).

Data set 5 (54 variables, tight but solvable)						
Search	Mean	Median	Mode	Standard dev.	Minimum	Maximum
BT	52	52	52	0	52	52
LS	42.88	44	46	3.94	29	50
ERA	53.99	54	54	0.04	53	54
RDGR	52.17	52	52	0.78	50	54
RGR	51.70	52	52	1.04	49	54

Table 5. Statistics of solution size (500 runs, 10 min each).

Superiority of RDGR over ERA on over-constrained problems: On over-constrained problems (Figures 4, 15, and 16 and Table 3), the deadlock phenomenon prevents ERA from finding solutions of quality comparable to those found by the other techniques [9–11]. BT, LS, RDGR, and RGR do not exhibit such a dichotomy of behavior between over-constrained cases and solvable instances.

Performance of ERA: On solvable problem instances (Figures 8 and 12), ERA dominates all techniques. It is the only algorithm that finds complete solutions for nearly all the runs. ERA completely dominates LS. However, on over-constrained problem instances (Figures 4 and 16) RDGR, RGR, BT and LS are superior to ERA due to the deadlock phenomenon. At the phase transition (Figures 19 and 20), the behavior of ERA is independent of the solvability of the problem. ERA performs only better than LS, while RDGR, RGR and BT perform better than ERA. This difference in performance of ERA may have to do with the structure of the randomly generated problems and the GTA problem. More tests are needed to understand this phenomenon.

RDGR is more stable than RGR: Due to their stochastic nature, RDGR and RGR techniques show a high instability in their solution quality. However, the standard deviation column of Tables 4 and 5 show that RDGR is relatively more stable than RGR.

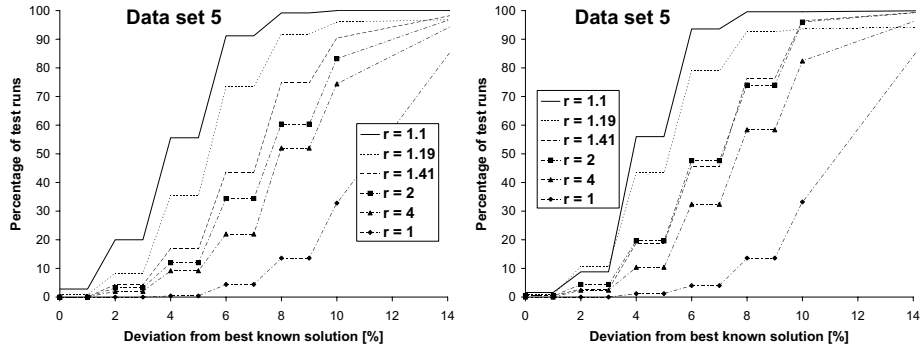


Fig. 10. SQDs of RGR with different ratios for **Fig. 11.** SQDs of RDGR with different ratios for cutoff value (500 runs, 5 min each).

Sensitivity of LS to local optima: LS sensitivity to local optima makes it particularly unattractive. Even BT outperforms LS.

Larger number of restarts in RDGR: On data set 1, the average number of restarts is 74.5 for RDGR and 16.7 of RGR. On data set 5, the average number of restarts is 56.9 for RDGR and 22.4 for RGR. This confirms our expectations stated in Section 3.2 that RDGR performs more restarts than RGR.

The following three statements, where \succ denotes an algorithm dominance over another, summarize the behavior of the 5 search strategies, also shown in Table 6:

- On solvable instances: ERA \succ RDGR \succ RGR \succ BT \succ LS
- On over-constrained instances: RDGR \succ RGR \succ BT \succ LS \succ ERA
- At the phase transition: RDGR \succ RGR \succ BT \succ ERA \succ LS

5 Conclusions and future work

By addressing a real-world application, we are able to identify, characterize, and compare the behavior of various search techniques. While BT is stable, it suffers from thrashing. LS is vulnerable to local optima. ERA shows difference in performance with different problem types. ERA has an amazing ability to solve under-constrained problems. However, ERA's performance degrades on over-constrained problems due to the deadlock phenomenon. This same deadlock phenomenon may be affecting ERA at the phase transition. Restart strategies effectively prevent thrashing, but their solution quality is highly variable. RGR operates by increasing cutoff values at every restart, which makes it more increasingly vulnerable to thrashing. RDGR attenuates this effect by making the cutoff value depend upon the result obtained at the previous restart, thus increasing the number of restarts in comparison to RGR. Consequently, RDGR exhibits a more stable behavior than RGR while yielding at least as good solutions. In the future, we plan to study the following directions:

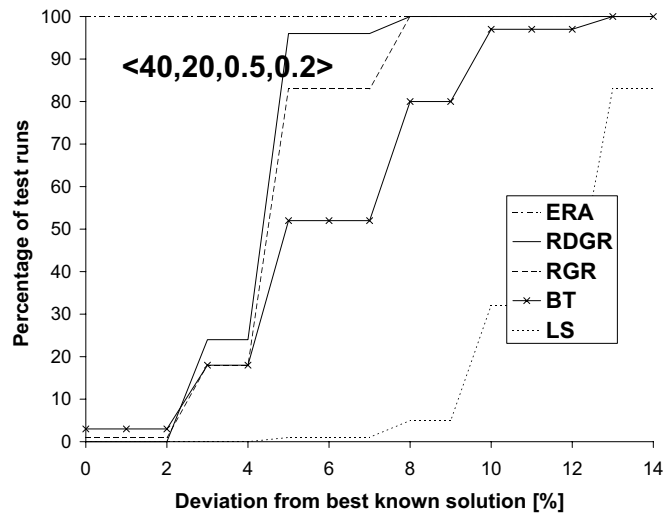


Fig. 12. SQDs of data set R1 (100 instances, 3 min each).

1. Validate our findings on other real-world case-studies. And,
2. Design new search hybrids where a solution from a given technique such as ERA is fed as a seed to another one such as heuristic backtrack search.

Acknowledgments. This work is supported by NSF grants #EPS-0091900 and CAREER #0133568. The experiments were conducted utilizing the Research Computing Facility of the University of Nebraska-Lincoln.

References

1. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98), Madison, Wisconsin (1998) 431–437
2. Walsh, T.: Search in a small world. In: Proc. of the 16th IJCAI. (1999) 1172–1177
3. Glaubius, R.: A Constraint Processing Approach to Assigning Graduate Teaching Assistants to Courses. Undergraduate Honors Thesis. Department of Computer Science & Engineering, University of Nebraska-Lincoln (2001)
4. Glaubius, R., Choueiry, B.Y.: Constraint Modeling and Reformulation in the Context of Academic Task Assignment. In: Working Notes of the Workshop Modelling and Solving Problems with Constraints, ECAI 2002, Lyon, France (2002)
5. Glaubius, R., Choueiry, B.Y.: Constraint Modeling and Reformulation in the Context of Academic Task Assignment. Poster presentation at the Fifth International Symposium on Abstraction, Reformulation and Approximation (SARA 2002) (2002)
6. Glaubius, R., Choueiry, B.Y.: Constraint Modeling in the Context of Academic Task Assignment. In Hentenryck, P.V., ed.: 8th International Conference on Principle and Practice of Constraint Programming (CP'02). Volume 2470 of LNCS., Springer (2002) 789

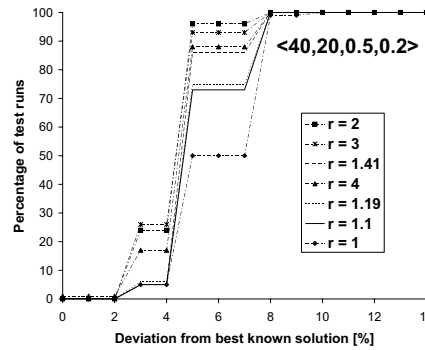
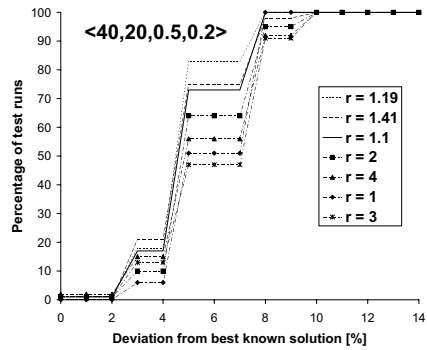


Fig. 13. RGR with different ratios for data set R1 **Fig. 14.** RDGR with different ratios for data set R1 (100 instances, 3 min each).

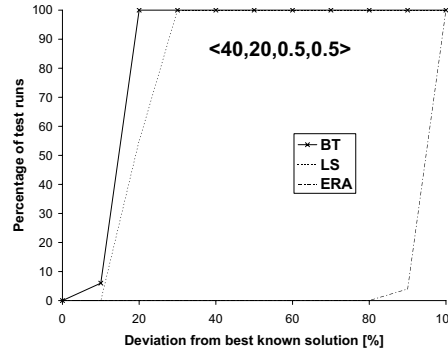
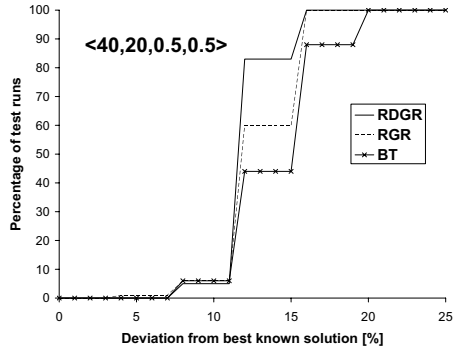


Fig. 15. SQDs of RDGR, RGR, and BT for data set R2 (100 instances, 3 min each). **Fig. 16.** SQDs of BT, LS, ERA for data set R2 (100 instances, 3 min each).

7. Freuder, E.C., Wallace, R.J.: Partial Constraint Satisfaction. *Artificial Intelligence* **58** (1992) 21–70
8. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* **9** (3) (1993) 268–299
9. Zou, H., Choueiry, B.Y.: Characterizing the Behavior of a Multi-Agent Search by Using it to Solve a Tight, Real-World Resource Allocation Problem. In: *Workshop on Applications of Constraint Programming*, Kinsale, County Cork, Ireland (2003) 81–101
10. Zou, H.: Iterative Improvement Techniques for Solving Tight Constraint Satisfaction Problems. Master’s thesis, Department of Computer Science & Engineering, University of Nebraska-Lincoln (2003)
11. Zou, H., Choueiry, B.Y.: Multi-agent Based Search versus Local Search and Backtrack Search for Solving Tight CSPs: A Practical Case Study. In: *Working Notes of the Workshop on Stochastic Search Algorithms (IJCAI 03)*, Acapulco, Mexico (2003) 17–24
12. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* **58** (1992) 161–205

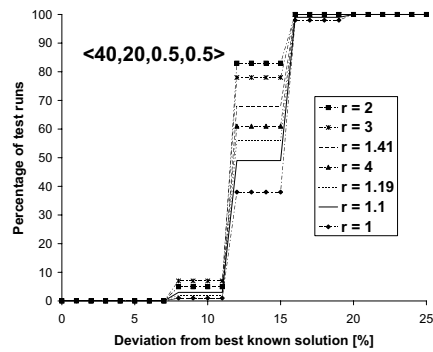
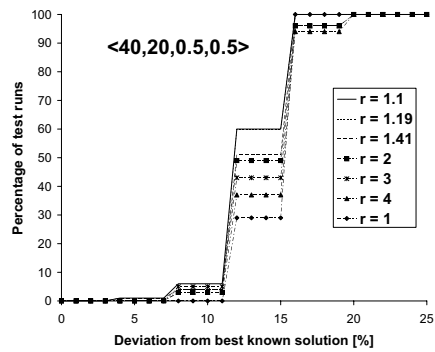


Fig. 17. RGR with different ratios for data set R2 (100 instances, 3 min each). **Fig. 18.** RDGR with different ratios for data set R2 (100 instances, 3 min each).

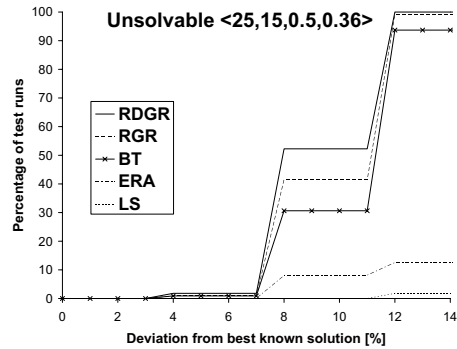
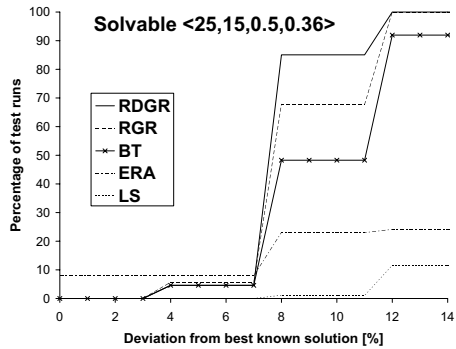


Fig. 19. SQD for the data set R3 (100 instances, 3 min each). **Fig. 20.** SQD for the data set R4 (100 instances, 3 min each).

13. Barták, R.: On-Line Guide to Constraint Programming. kti.ms.mff.cuni.cz/~bartak/constraints (1998)
14. Liu, J., Jing, H., Tang, Y.: Multi-agent oriented constraint satisfaction. *Artificial Intelligence* **136** (2002) 101–144
15. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. In: *Israel Symposium on Theory of Computing Systems*. (1993) 128–133
16. Hoos, H., Stützle, T.: *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann (2004) Forthcoming.
17. van Hemert, J.I.: RandomCSP: generating constraint satisfaction problems randomly. homepages.cwi.nl/~jvhemert/randomcsp.html (2004)

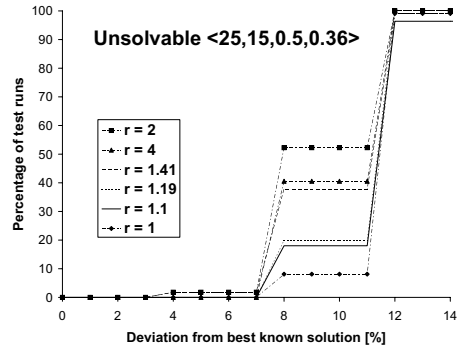
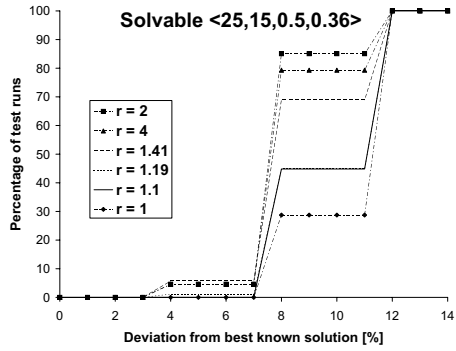


Fig. 21. SQDs of RDGR with different ratios for cutoff value (100 instances, 3 min each).

Fig. 22. SQDs of RDGR with different ratios for cutoff value (100 instances, 3 min each).

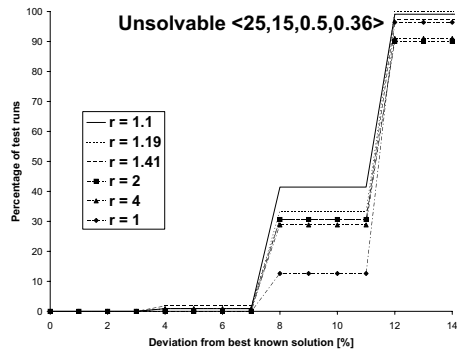
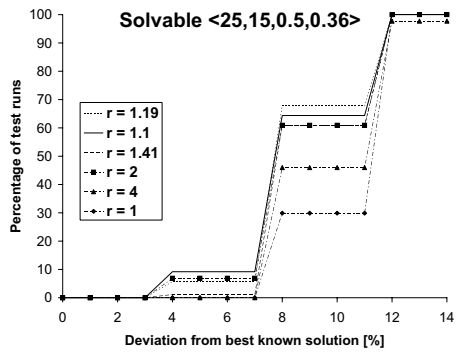


Fig. 23. SQDs of RGR with different ratios for cutoff value (100 instances, 3 min each).

Fig. 24. SQDs of RGR with different ratios for cutoff value (100 instances, 3 min each).

	Characteristics
ERA	General: Stochastic and incomplete
	Tight but solvable problems: Immune to local optima
	Over-constrained problems: Deadlock causes instability and yields shorter solutions
LS	General: Stochastic, incomplete, and quickly stabilizes
	Tight but solvable problems: Liable to local optima, and fails to solve tight CSPs even with random-walk and restart strategies
	Over-constrained problems: Finds longer solutions than ERA
RDGR	General: Stochastic, incomplete, immune to thrashing, produces longer solutions than BT, immune to deadlock, reliable on unknown instances, and immune to local optima, but less than ERA
RGR	General: Stochastic, Approximately complete, less immune to thrashing than RDGR, and yields shorter solutions than RDGR in general.
BT	General: Systematic, complete (theoretically, rarely in practice), liable to thrashing, yields shorter solutions than RDGR and RGR, stable behavior, and more stable solutions than stochastic methods in general

Table 6. Comparing the behaviors of search strategies in our context.

Desk-mates (Stable Matching) with Privacy of Preferences, and a new Distributed CSP Framework

Marius-Călin Silaghi¹ and Markus Zanker² and Roman Barták³

¹ Florida Institute of Technology, USA

² Universität Klagenfurt, Austria

³ Charles University, Czech Republic

msilaghi@fit.edu, markus@ifit.uni-klu.ac.at, bartak@kti.mff.cuni.cz

Abstract. The desk-mates matcher application solves the need of placing students in pairs of two for working in projects (need that is similar to the well known problems of stable matchings or stable roommates). Each of the persons in the previous application has a (hopefully stable) secret preference between every two possible partners. The participants want to find an allocation satisfying their secret preferences and without leaking any of these secret preferences, except for what a participant can infer from the identity of the partner that was recommended to her.

The peculiarities of this problem requires solvers based on old distributed CSP frameworks to use models whose search spaces are higher than those in centralized solvers, with bad effects on efficiency.

We introduce a distributed weighted constraint satisfaction (DisWCSP) framework where the actual constraints are secrets that are not known by any agent. They are defined by a set of functions on some secret inputs from all agents. The solution is also kept secret and each agent learns just the result of applying an agreed function on the solution. The new framework is shown to improve the efficiency ($O(2^{m^3 - \log(m)})$ times) in modeling and solving the aforementioned problem with m participants. We show how to extend our previous techniques to solve securely problems modeled with the new formalism, and exemplify with the problem in the title. An applet-based solver is available [Sil04a].

1 Introduction

The desk-mates matcher application groups a set of students in stable working teams of two, such that whenever one person wants to change her partner for a third one, the third one prefers her current partner to the change (similar to stable matchings or stable roommates [IM02]). The students have a secret preference between any pair of potential partners, and between working with any given partner or working alone.

Versions of these problems, without privacy requirements, have been long known and studied. It is an example of constraint satisfaction problem (CSP) [GP02].¹ A CSP is described by a set of variables and a set of constraints on the possible values of those variables. The CSP problem consists in finding assignments for those variables with values from their domains such that all constraints are satisfied. The centralized CSP

¹ Operations research has provided very efficient solutions to some instances without privacy.

techniques require every eventual participant to reveal its preferences (e.g. to a trusted server), to compute the solution. Therefore, they apply only when the participants accept to reveal their preferences to the trusted party.

There exist frameworks and techniques to model and solve distributed CSPs (DisCSPs) with privacy requirements, namely when the domains of the variables are private to agents [YDIK98,MJ00], or when the constraints are private to agents [SSHF00a,Sil03b,SR04].

However, the desk-mates problem seem not to be modeled efficiently (i.e. with a reduced search space) with any of the two known types of distributed CSP frameworks. In this article we propose a new framework for the distributed constraint satisfaction problems. It can model naturally existing distributed constraint satisfaction problems, and also the desk-mates (stable matchings problems). The new framework assumes that the constraints are not known to absolutely any agent but they are computable from secret inputs, by applying public functions on them. These functions use secret inputs provided securely by the different participants. Similarly, the final assignments are secret and each agent can retrieve just the result of applying some agreed function on the secret solution.

We also show how secure multi-party computation techniques that we have recently developed for solving DisCSPs with private constraints can be extended to solve problems described in the new framework. We start introducing formally the CSP problem.

CSP. A constraint satisfaction problem (CSP) is defined by three sets: (X, D, C) . $X = \{x_1, \dots, x_m\}$ is a set of variables and $D = \{D_1, \dots, D_m\}$ is a set of finite domains such that x_i can take values only from $D_i = \{v_1^i, \dots, v_{d_i}^i\}$. $C = \{\phi_1, \dots, \phi_c\}$ is a set of constraints. A constraint ϕ_i limits the legality of each combination of assignments to the variables of an ordered subset X_i of the variables in X , $X_i \subseteq X$. An assignment is a pair $\langle x_i, v_k^i \rangle$ meaning that the variable x_i is assigned the value v_k^i .

A tuple is an ordered set. The projection of a tuple ϵ of assignments over a tuple of variables X_i is denoted $\epsilon|_{X_i}$. A solution of a CSP (X, D, C) is a tuple of assignments, ϵ^* , with one assignment for each variable in X such that each $\phi_i \in C$ is satisfied by $\epsilon^*|_{X_i}$. The search space of a CSP is the Cartesian product of the domains of its variables.

We consider that a set of participants are the source of such CSPs and one has to find agreements for a solution, from the set of possible alternatives, that satisfies a set of (secret) requirements of the participants. This view suggests a concept of a distributed CSP. Several frameworks were proposed so far for Distributed Constraint Satisfaction [ZM91,CDK91,YSH02a,MJ00]. Some versions consider that each agent owns a constraint of the CSP [ZM91,SGM96]. This constraint could model the private information of the agent [SSHF00a]. Other versions consider that each agent owns the domain of a variable while the constraints are shared [YDIK98]. The secret domains can also model some private constraints of the agent.

None of the two approaches, namely private variables or private domains, can model efficiently the stable matching problems. This is because the private data of these problems does not *directly* constrain the allocation of the natural shared resources (the matching). An indirect relation exist with such a constraint. Redundant variables would need to be introduced in the system, modeling the secret preferences, but reducing ef-

iciency. A new framework will be introduced in this article to avoid these redundant variables.

2 The Desk-mates Matcher Application

In some of our classes students are grouped in teams of two, for solving laboratory exercises as well as for working on projects. It is desirable for these teams to be stable for the duration of the project. Otherwise discontinuities and changes may reduce the efficiency of the learning process. Some students insist to work alone, and it is typically difficult for students to refuse other's offers of partnership. In fact students sometimes prefer to keep private their preferences between colleagues, to avoid hurting others. We decided that it is needed to provide students with a support in solving this situations. We therefore built a web-application that insures their privacy using cryptographic solvers of distributed constraint satisfaction problems, as proposed in this paper.

Our web-application works as follows. An organizer of the computation, e.g. an instructor or a student, uses the web form at [Sil04a] to generate (for the included JAVA applets) parameters that are customized for the computation at hand. This process requires the organizer to input the size of the class, the names of the students, and a cryptographic public Paillier key provided by each student. Students can generate Paillier key pairs using the corresponding applet linked from the form, and keep the secret keys while handing the private ones to the organizer.

When the customized problem description is generated, a website is automatically built and provided for this problem instance. The organizer is offered an opportunity to email its URL to the students. The organizer can also specify which algorithm to be used for the computation.

Each student browses the received URL, and downloads the applet with customized parameters. The browser can verify the integrity of the applet. Each student provides the applet with his secret key, and inputs his secret preferences. Then he launches his applet into the computation. The applets retrieve each-other's network IP number and port by using a directory server installed on the same host as the web-application. The applets solve the problem securely, and display for each student only the name of her/his partner.

The Distributed Configurator Application Our approach can also be applied to the problem of distributed configuration of products based on components from several providers, with secret configuration requirements.

3 Background

Our techniques here apply only to problems whose constraints and outputs can be represented as first order logic expressions, or as arithmetic circuits on inputs. Actually, we propose a procedure to translate first order logic definitions of constraints/outputs into arithmetic circuits. In the following we introduce arithmetic circuits and a short overview of the literature and techniques that made them relevant.

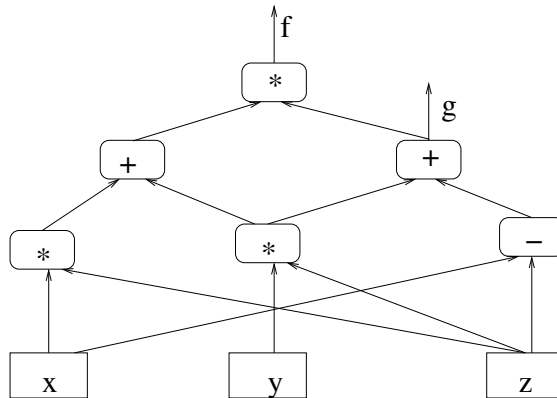


Fig. 1. An arithmetic circuit, $g = yz + (x - z)$ and $f = (xz + yz)g$. Each input can be the secret of some participant. The output may not be revealed to all participants. All intermediary values remain secret to everybody.

3.1 Secure Arithmetic Circuit Evaluation

Secure multi-party computations can simulate any arithmetic circuit [BOGW88] or boolean circuit [Kil88, Gol04] evaluation. An *arithmetic circuit* can be intuitively imagined as a directed graph without cycles where each node is described either by an addition/subtraction or by a multiplication operator (see Figure 1). Each leaf is a constant. In a secure arithmetic circuit evaluation, a set of participants perform the operations of an arithmetic circuit over some inputs, each input being either public or an (encrypted/shared) secret of one of them. The result of the arithmetic circuit are the values of some predefined nodes. The protocol can be designed to reveal the result to only a subset of the agents, while none of them learns anything about intermediary values. One says that the multi-party computation *simulates* the evaluation of the arithmetic circuit. A *boolean circuit* is similar, just that the leaves are boolean truth values, false or true, often represented as 0 and 1. The rest of the nodes are boolean operators like AND or XOR. A function does not have to be represented in this form to be solvable using general secure arithmetic circuit evaluation. It only needs to have such an equivalent representation. For example, the operation $\sum_{i=B}^E f(i)$ is an arithmetic circuit if B and E are public constants and $f(i)$ is an arithmetic circuit. The same is true about $\prod_{i=B}^E f(i)$. Such constructs are useful when designing arithmetic circuits.

There must be some machinery to compute the result of the circuit from the inputs. However, existing techniques allow for the secret inputs not to be revealed to this machinery. Namely the machinery works only with encrypted secrets that it cannot decrypt (i.e. using Shamir's secret sharing [Sha79], detailed later).

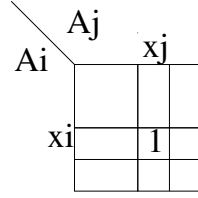


Fig. 2. A constraint between x_i and x_j for the desk-mates problems. An element is feasible, value '1', if the corresponding pairs (A_i, A_{x_i}) and (A_j, A_{x_j}) are allowed in a solution given the preferences of $A_i, A_j, A_{x_i}, A_{x_j}$.

4 Distributed CSPs with constraints secret to everybody

In this article we redefine the distributed CSP framework, aiming to model efficiently (i.e. with a reduced search space) the distribution of some famous CSP problems, namely the stable machings problems (e.g. the desk-mates problem).

Desk-mates The desk-mates problem consists in placing a set of persons $A = \{A_1, \dots, A_m\}$ in teams of two (or two-seats desks), such that if any person A_i prefers a person A_j to the desk-mate selected for her, then A_j prefers her current desk-mate to A_i .

A way of modeling the desk-mates problem as a CSP is to have one variable x_i for each person A_i specifying the index of the desk-mate assigned to her by the solution, or specifying i , the index of A_i itself, if she remains alone. The constraints are obtained by preprocessing the input from participants about their preferences. The fact that a person A_i prefers A_u to A_v is specified by the first order logic predicate $P_{A_i}(u, v)$. There is a constraint ϕ^{ij} between every pair of distinct variables x_i and x_j . In first order logic notation, the constraint between each two variables x_i and x_j is:

$$\forall x_i, x_j : \phi^{ij}(x_i, x_j) \stackrel{\text{def}}{=} (P_{A_i}(x_j, x_i) \Rightarrow P_{A_{x_j}}(j, i)) \wedge (P_{A_j}(x_i, x_j) \Rightarrow P_{A_{x_i}}(i, j)) \wedge ((x_i = j) \Leftrightarrow (x_j = i)) \quad (1)$$

Read: For each pair of participants A_i, A_j , (and corresponding variables x_i and x_j) there is a constraint ϕ^{ij} that allows a pair of assignments to these variables only if:

- the fact that A_i prefers the participant assigned to A_j (A_{x_j}) to her own match A_{x_i} implies that:
the agent assigned by these assignments to A_j (A_{x_j}), prefers the agent A_j to the agent A_i .
- the fact that A_j prefers the participant assigned to A_i (A_{x_i}) to her own match A_{x_j} implies that
the agent assigned by these assignments to A_i (A_{x_i}), prefers the agent A_i to the agent A_j .
- A_j is the match of A_i only if A_i is the match of A_j .

Note that this model subsumes the constraints: $\forall i, j : x_i \neq x_j$. The main complication with this kind of CSPs is that the constraints are functions of secrets that cannot be easily elicited from the participants. Distributed CSP frameworks are meant to address such problems.

Modeling the desk-mates problem with DisCSPs with secret constraints that are known to some agents. One can model the desk-mates problem with secret constraints known to some agents [ZM91,SSH00b] by choosing as variables, x_1, \dots, x_m , the index of the partner associated to each agent (that has to be computed) and using one additional boolean variable for each secret preference, $P_{A_i}(u, v)$. The total number of boolean variables is m^3 , m^2 of them being actually fixed by public constraints (e.g. $P_{A_i}(u, u) = 0$). However, also taking into account the variables x_1, \dots, x_m , the total search space becomes $O(m^m 2^{m^3})$. This is $O(2^{m^3})$ times worse than the centralized CSP formalization whose search space is only $O(m^m)$.

We propose now a distributed constraint satisfaction framework that allows to model these problems with the same search space size as the CSP framework, $O(m^m)$.

4.1 Redefining the Distributed Constraint Satisfaction Framework

In the previous part of this section we have exemplified CSP models for the stable matchings problem. We have seen that it is difficult to model efficiently these problems using existing private variable-, or private constraint- oriented distributed constraint satisfaction frameworks.

Let us propose a framework for modeling distributed CSPs, where a constraint is not (necessarily) a secret known to an agent, or public, but can also be a secret unknown to all agents.

Any distributed problem is essentially (in our view) described by a set of inputs and expected outputs from/to each participant. A distributed CSP is a specialization in the sense that the inputs are used to specify constraints/domains of a CSP, and the outputs are derived from the solution of that CSP. As shown elsewhere, sometime the inputs are also needed (in combination with the solution) to provide meaningful outputs [Sil04b].

Definition 1. *A Distributed CSP (DisCSP) is defined by six sets (A, X, D, C, I, O) and an algebraic structure F . $A = \{A_1, \dots, A_n\}$ is a set of agents. X, D , and the solution are defined like for CSPs.*

$I = \{I_1, \dots, I_n\}$ is a set of secret inputs. I_i is a tuple of α_i secret inputs (defined on F) from the agent A_i . Each input I_i belongs to F^{α_i} .

Like for CSPs, C is a set of constraints. There may exist a public constraint in C , ϕ_0 , defined by a predicate $\phi_0(\epsilon)$ on tuples of assignments ϵ , known to everybody. However, each constraint $\phi_i, i > 0$, in C is defined as a set of known predicates $\phi_i(\epsilon, I)$ over the secret inputs I , and the tuples ϵ of assignments to all the variables in a set of variables $X_i, X_i \subseteq X$.

$O = \{o_1, \dots, o_n\}$ is the set of outputs to the different agents. Let m be the number of variables. $o_i : D_1 \times \dots \times D_m \rightarrow F^{\omega_i}$ is a function receiving as parameter a solution and returning ω_i secret outputs (from F) that will be revealed only to the agent A_i .

Theorem 1. *The framework in the Definition 1 can model any distributed constraint satisfaction problems with private constraints [SSHF00b].*

Proof. The new DisCSP framework can be used to model any of the DisCSP problems with constraints private to agents, by defining I_i as the extensional representation of the private constraint of A_i (assuming the simple but sufficient case of one constraint per agent). $\phi_i(\epsilon, I)$ is then given by the corresponding value for ϵ in I_i (true/1 or false/0). The outputs are going to be $o_i(\epsilon) = \epsilon$ for all i . q.e.d.

Theorem 2. *The framework in the Definition 1 can model distributed constraint satisfaction problems with private domains [YDIK98].*

Proof. A private domain of an agent can also be modeled as a private unary constraint, in a DisCSP where each domain is the maximum possible domain for the variable. Then, the Theorem 1 applies. q.e.d.

We do not claim that the new framework is more general than the existing frameworks. It enables us to model naturally and efficiently the desk-mate (stable matchings) problems. One can also model these problems with the old frameworks, but they seem to yield much larger search spaces, and therefore less efficient solutions. Let us now exemplify how this framework can model the new problems.

Modeling the desk-mates problem as a DisCSP. A way of modeling the desk-mates problem as a DisCSP is to have one agent, A_i , and one variable, x_i , for each participant in the problem description. x_i specifies the index of the desk-mate assigned to A_i by the solution, or specifies i if she remains alone. The inputs I_i of each agent are given by the set of preferences $P_{A_i}(u, v)$, specifying whether A_i prefers A_u to A_v , for each u and v . The set F , to which belong the inputs and the outputs, is $\{true, false\}$.

There is a constraint ϕ^{ij} between every pair of variables x_i and x_j , defined as in Equation 1. The output functions are defined as: $o_i(\epsilon) \stackrel{\text{def}}{=} \epsilon_{\{x_i\}}$. Namely, each agent learns only the name of her desk-mate. There is a public constraint:

$$\phi_0 \stackrel{\text{def}}{=} \forall i, j, ((x_i = j) \Leftrightarrow (x_j = i)) \wedge (x_i \neq x_j) \quad (2)$$

4.2 Distributed Weighted Constraint Satisfaction Problems

Definition 2. *A distributed constraint satisfaction problem (DisWCSP) is defined by six sets (A, X, D, C, I, O) , and algebraic structure F , and a set of acceptable solution qualities B , that can be often represented as an interval $[B_1, B_2]$.*

- $A = \{A_1, \dots, A_n\}$ is a set of agents.
- $X = \{x_1, \dots, x_m\}$ is a set of variables and $D = \{D_1, \dots, D_m\}$ is a set of finite domains such that x_i can take values only from $D_i = \{v_1^i, \dots, v_{d_i}^i\}$. An assignment is a pair $\langle x_i, v_k^i \rangle$ meaning that the variable x_i is assigned the value v_k^i . A tuple is an ordered set.
- $I = \{I_1, \dots, I_n\}$ is a set of secret inputs. I_i is a tuple of α_i secret inputs (defined on a set F) from the agent A_i . Each input I_i belongs to F^{α_i} .

- $C = \{\phi_0, \dots, \phi_c\}$ is a set of constraints. A constraint ϕ_i weights the legality of each combination of assignments to the variables of an ordered subset X_i of the variables in X , $X_i \subseteq X$. ϕ_0 is a public constraint defined by a function $\phi_0(\epsilon)$ on tuples of assignments ϵ , known to everybody. Each constraint ϕ_i , $i > 0$, in C is defined as a known function $\phi_i(\epsilon, I)$ over the secret inputs I , and the tuples ϵ of assignments to all the variables in a set of variables X_i , $X_i \subseteq X$. $\phi_i(\epsilon, I)$ maps secret inputs and tuples into weights.
- The projection of a tuple ϵ of assignments over a tuple of variables X_i is denoted $\epsilon|_{X_i}$. A solution is $\epsilon^* = \underset{\epsilon \in D_1 \times \dots \times D_n}{\operatorname{argmin}} \sum_{i=0}^c \phi_i(\epsilon|_{X_i})$, if $\sum_{i=0}^c \phi_i(\epsilon^*|_{X_i}) \in [B_1 \dots B_2]$.
- $O = \{o_1, \dots, o_n\}$ is the set of outputs to the different agents. $o_i : I \times D_1 \times \dots \times D_m \rightarrow F^{\omega_i}$ is a function receiving as parameter the inputs and a solution, and returning ω_i secret outputs (from F) that will be revealed only to the agent A_i .

Solvers developed in our previous work require that the functions in sets O and C are input either in first order logic form, or in the form of arithmetic circuits.

The inputs are not revealed to the solving machinery, as it manipulates only encrypted data (in difference from a classic monolithic system with a trusted server).

The public constraint ϕ_0 can be input into the system using a set of constraints $\{\phi_0^1, \phi_0^2, \dots\}$, and the tuples of assignments accepted by ϕ_0 can be obtained separately by each agent, when needed, using any systematic search technique that finds all solutions of a CSP, e.g. backtracking or lookahead algorithms (BT, BM, CBJ, FC, MAC, EMAC, etc.).

5 Adapting existing secure solvers to the new DisCSP framework

There exist a large set of algorithms addressing distributed CSPs with privacy of constraints [Sil02, HCN⁺01, FMW01, WS04, YSH02b, Sil03b]. Note that none of the existing techniques involves propagation, except for a very old variant in [Sil02]. The ones that we succeed to extend to the new framework are:

- Finding the set of all solutions of a distributed constraint problem with secret constraints [HCN⁺01].
- Finding the first solution in a lexicographic order for a distributed constraint satisfaction problem with secret constraints that are known to some agents [Sil03a].
- Finding a random solution for a DisCSP with secret constraints that are known to some agents [Sil03b].

When a solution is returned to the desk-mates problem, each agent A_i can infer that: *any agent A_k preferred by A_i to her current desk-mate A_j , prefers her current partner to A_i .* If only one solution is returned (picked randomly among the existing solutions), then no other secret preference can be inferred with certainty.

Theorem 3. *The desk-mates problem can have several solutions.*

Proof. Consider a case with three agents, A_1, A_2, A_3 where $P_{A_1}(2, 3), P_{A_2}(3, 1), P_{A_3}(1, 2)$. This is a loop of preferences, and has three stable solutions, the sets of teams $\{(A_1, A_2), (A_3)\}, \{(A_2, A_3), (A_1)\}, \{(A_3, A_1), (A_2)\}$. Such an example can be constructed out of any similar loop of preferences, of any size.

If there exist several solutions, the agents will prefer not to reveal more than one of them. The remaining solutions would only reveal more secret preferences:

- Typically there is no other fair way, except randomness, to break the tie between several solutions.
- If the single solution that is returned is selected as the first one in some given lexicographic order on the variables and domains of the problem, then additional information is leaked concerning the fact that tuples placed lexicographically before the suggested solution do not satisfy the constraints [Sil03b].

As it follows, if it is known that a certain problem has only one solution, then any technique is acceptable among either:

- Finding and returning all solutions using the technique in [HCN⁺01], or
- Returning only the first solution (e.g. by sequentially checking each tuple in lexicographical order until a solution is found).

Otherwise, strong privacy requirements make techniques returning a random solution [Sil03b] desirable, despite their potential of having a lower efficiency.

5.1 General Scheme

We will note that the main difference between the new DisCSP framework, and the one with secret constraints that are known to some agents, is that now the constraints need to be computed dynamically from secrets inputs. All the techniques we extend to the new framework contain a component based on Shamir’s secret sharing [Sha79]. It is the achievement of this sharing which is most affected by the change in framework. We will start by describing Shamir’s secret sharing, its importance in distributed multi-party computations, and then we will introduce our changes.

The secure multi-party simulation of arithmetic circuit evaluation proposed in [BOGW88] exploits Shamir’s secret sharing [Sha79]. This sharing is based on the fact that a polynomial $f(x)$ of degree $t-1$ with unknown parameters can be reconstructed given the evaluation of f in at least t distinct values of x , using Lagrange interpolation. Absolutely no information is given about the value of $f(0)$ by revealing the valuation of f in any at most $t-1$ non-zero values of x . Therefore, in order to share a secret number s to n participants A_1, \dots, A_n , one first selects $t-1$ random numbers a_1, \dots, a_{t-1} that will define the polynomial $f(x) = s + \sum_{i=1}^{t-1} (a_i x^i)$. A distinct non-zero number τ_i is assigned to each participant A_i . The value of the pair $(\tau_i, f(\tau_i))$ is sent over a secure channel (e.g. encrypted) to each participant A_i . This is called a (t, n) -threshold scheme. Once secret numbers are shared with a (t, n) -threshold scheme, evaluation of an arbitrary arithmetic circuit can be performed over the shared secrets, in such a way that all results remain shared secrets with the same security properties (the

number of supported colluders, $t-1$) [BOGW88, Yao82]. For [Sha79]’s technique, one knows to perform additions and multiplications when $t \leq (n-1)/2$. Since any $\lfloor n/2 \rfloor$ participants cannot find anything secret by colluding, such a technique is called $\lfloor n/2 \rfloor$ -private [BOGW88].

We do not try to encode functions, but only their inputs. All functions (more exactly, arithmetic circuits) that will be computed are public and known by all participants. Their inputs, intermediary values, and outputs are shared secrets. The functions that we are able to compute belong to the class of arithmetic circuits. The techniques computing these function do not reveal any information to anybody, and work by letting agents processing the Shamir shares that they know, and by sharing additional secret values.

The techniques solving DisCSPs with private constraints can be used as a black box, except for the secret constraint sharing. Namely, instead of simply sending encrypted Shamir shares [Sha79] of one’s constraint, those shares of the constraints have to be computed from the secret inputs of the agents. We therefore propose to replace the secret sharing/reconstruction steps with simulations of arithmetic circuit evaluation which will compute each $\phi_k(\epsilon, I)$ for each tuple ϵ and for the actual inputs I . This step is called *preprocessing*. Intuitively, preprocessing is the step of computing the encrypted initial parameters of the CSP (i.e. acceptance/feasibility value of a tuple from the point of view of each constraint), out of the provided secret inputs. Preprocessing prepares “the pairs” $(y, f(y))$ that encode the 0/1 values of the constraints. It is done by evaluating arithmetic circuits.

Similarly, instead of just reconstructing the assignments to variables in a solution ϵ , one will have to design and execute secure computations of the functions $o_k(\epsilon)$. This step is called *post-processing*. Intuitively, post-processing is the step of computing the outputs to be revealed to agents, from the obtained encrypted solution of the DisCSP and secret inputs. We show that in our cases this can also be done using simulations of arithmetic circuit evaluations.

Assume \mathcal{A} is some algorithm using Shamir’s secret sharing for securely finding a solution of a distributed CSP (with secret constraints known to some agents). The generic extension of the algorithm \mathcal{A} to solve the DisCSP in the new framework is:

- **Preprocessing:** Share the secrets in I with Shamir’s secret sharing scheme. Compute each $\phi_k(\epsilon_{|X_k}, I)$ for each tuple $\epsilon_{|X_k}$ and for the actual inputs I by designing it as an arithmetic circuit and simulating securely its evaluation. The public constraint ϕ_0 can be shared by any agent.
- Run the algorithm \mathcal{A} as a black-box, for finding a solution ϵ^* shared with Shamir’s secret sharing scheme, for a DisCSP with parameters (i.e. constraints) shared with Shamir’s secret sharing scheme.
- **Post-processing:** Compute each $o_i(\epsilon^*)$ by designing it as an arithmetic circuit and simulating securely its evaluation. Reveal the result of $o_i(\epsilon^*)$ only to A_i .

5.2 Pre- and post- processing for desk-mate problems

In the remaining part of the article we will prove that it is possible to design the needed preprocessing and post-processing to solve our example of DisCSPs, the desk-mates problem, using the general scheme defined above.

Preprocessing for the desk-mates problem. We assume the same choice of variables, as for the CSP formalization of this problem in Section 4. Let us now show how simple arithmetic circuits can implement the required preprocessing.

Each variable x_i specifies the index of the desk-mate associated to A_i . The input of each agent A_i is a preference value $P_{A_i}(j, k)$ for each ordered pair of agents (A_j, A_k) , and specifying whether A_i prefers A_j to A_k . $P_{A_i}(j, k)=1$ if and only if A_i prefers A_j to A_k . Otherwise $P_{A_i}(j, k)=0$. A constraint $\phi^{i,j}$ is defined between each two variables, x_i and x_j . I.e. $\phi^{i,j}[u, v]$ is the acceptance value of the pair of matches: $(A_i, A_u), (A_j, A_v)$. One synthesizes $m(m-1)/2$ such constraints:

$$\phi^{i,j}[u, v] = \begin{cases} 0 & \text{when } u = v \\ (1 - P_{A_i}(v, u) * (1 - P_{A_v}(j, i))) * \\ (1 - P_{A_j}(u, v) * (1 - P_{A_u}(i, j))) & \text{when } u \neq v \end{cases}$$

The public constraint ϕ_0 (same as in Equation 2) restricts each pair of assignments:

$$\forall \epsilon, \epsilon = (\langle x_i, u \rangle, \langle x_j, v \rangle) : \phi_0(\epsilon) \stackrel{\text{def}}{=} ((u=j) \Leftrightarrow (v=i)) \wedge (u \neq v)$$

ϕ_0 is known by everybody, and therefore there is no need to compute it with arithmetic circuits. The complexity of this preprocessing is $O(m^4)$ multiplications of secrets (for m^2 binary constraints with m^2 tuples each).

The desk-mates problem does not require any arithmetic circuit evaluation for the post-processing, as each agent A_i learns a value existing in the solution, $o_i(\epsilon) = \epsilon_{\{x_i\}}$. The participants just reveal to A_i their shares of x_i in the solution.

6 Transforming first order logic in arithmetic circuits

Based on the experience with the examples analyzed so far, we conclude that with the new DisCSP framework it is useful to have a mechanism for automatic translation of first order logic sentences about secrets, into arithmetic circuits.

The main constructs in first order logic whose translation to arithmetic circuits will be given here are: $\forall i \in [1..n]P(i)$, $\exists i \in [1..n]P(i)$, $P \wedge Q$, $P \vee Q$, $\neg P$, $\min_{P(i)}(i)$, and $f = k$, where P and Q are predicates with a true (1) or false (0) value, f is a secret integer in a given interval, $[1..n]$, i is a quantified variable that can take integer values in a given interval, $[1..n]$, and k is a constant. They can also apply to variables and secrets from any finite set of numbers, $S = \{a_1, \dots, a_n\}$. $\min_{P(i)} i$ is the function returning the minimum i such that $P(i)$ holds. The equivalent arithmetic circuits are shown in Table 1.

6.1 Complexity

For a problem with size of the search space Θ and c constraints, the number of messages for finding all solutions with secure techniques similar to the one in [HCN⁺01] is given by $(c-1)\Theta$ multiplications of shared secrets ($n(n-1)$ messages for each such multiplication). For the desk-mates problem modeled with the new framework, $\Theta=m^m$

First Order Logic Sentence	Equivalent Arithmetic Circuit
P	\overline{P}
$\forall i \in [1..n], P(i)$	$\prod_{i=1}^n \overline{P(i)}$
$\forall a \in S, P(a)$	$\prod_{i=1}^n \overline{P(a_i)}$
$\exists i \in [1..n], P(i)$	$\sum_{i=1}^n [P(i) \prod_{j=1}^{i-1} (1 - P(j))]$
$\exists a \in S, P(a)$	$\sum_{i=1}^n [P(a_i) \prod_{j=1}^{i-1} (1 - P(a_j))]$
$P \wedge Q$	$\overline{P * Q}$
$P \vee Q$	$\overline{P + (1 - P)Q}$
$P \Rightarrow Q$	$1 - P(1 - Q)$
$\neg P$	$1 - P$
$f = k, (f, k \in [1..n])$ (i.e. test if f equals k , where they are in $[1..n]$)	$\frac{1}{(k-1)!(n-k)!} \prod_{i=1}^{k-1} (f - i) \prod_{i=k+1}^n (i - f)$
$f = a_k, (f \in S, k \in [1..n])$ (i.e. test if f equals the k^{th} element of S)	$\frac{\prod_{a_i \in S, i \neq k} (f - a_i)}{\prod_{a_i \in S, i \neq k} (a_k - a_i)}$
$\min_{P(i), i \in [1..n]} i$ (i.e., smallest i s.t. $P(i)$ holds)	$\sum_{i=1}^n [i \overline{P(i)} \prod_{j=1}^{i-1} (1 - \overline{P(j)})]$

Table 1. Equivalences between first order logic constructs and arithmetic circuits. P and Q are predicates and \overline{P} and \overline{Q} are their equivalent arithmetic circuits. $S = \{a_1, \dots, a_n\}$.

and $c=1$ for the version with a single global constraint, or $c=m^2/2$ for the version with binary constraints. For the case with binary constraints, it yields a complexity of $O(m^{m+2})$. As mentioned before, the preprocessing has complexity $O(m^4)$ multiplications between shared secrets, resulting in a total complexity $O(m^2(m^m + m^2))$.

Solving the same problem with the same algorithm but modeled with the old DisCSP framework with private constraints, $\Theta = m^m 2^{m^3}$ and $c = m$, for one global constraint from each agent. There is no preprocessing, but the total complexity is $O(m^{m+1} 2^{m^3})$. The new framework behaves better since $m \ll 2^{m^3}$. The comparison is similar for other secure algorithms, like MPC-DisCSP1 (see [Sil03b]) whose complexity is given by $O(dm(c+m)\Theta)$ multiplications between shared secrets.

Similar improvements can be achieved by applying this new framework to other known problems like incentive auctions and stable marriages problems [Sil04b].

7 Conclusions

DisCSPs [BMM01,SGM96,LV97,Ham99,MR99,ZWW02,BD97,FBKG02,MTSY04] are a very active research area. Privacy has been recently stressed in [MJ00,FMW01,WF02,FMG02,YSH02b] as an important goal in designing algorithms for solving DisCSPs.

In this article we have investigated how versions of old and famous problems, stable matchings problems, can be solved such that the privacy of the participants is guaran-

teed except for what is leaked by the selected solution. Our approach uses secure simulations of arithmetic circuit evaluations and is therefore robust whenever no majority of the participants colludes to find the secret of the others, and when all agents follow the protocol.

We note that the desk-mates problems cannot be efficiently modeled (at least not in an obvious way) with existing distributed constraint satisfaction frameworks. We have therefore introduced a new distributed constraint satisfaction framework that can model such problems with the same search space size as the classic centralized CSP models. We have shown how some techniques for the existing frameworks can be adapted to problems modeled with the new DisCSPs, and we exemplify the model with the desk-mates problems. For m participants in the desk-mates problem, the size of the search space in the DisCSP model achieved with the new framework is $O(m^m)$ while the previous framework with private constraints yields DisCSP instances with a size of the search space of $O(m^m 2^{m^3})$. In existing secure algorithms for solving DisCSPs, the number of exchanged messages is fix and directly proportional to the search space size, making this property of a problem instance particularly relevant.

References

- [BD97] B. Baudot and Y. Deville. Analysis of distributed arc-consistency algorithms. Technical Report RR-97-07, U. Catholique Louvain, 1997.
- [BMM01] C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *CP*, page 772, 2001.
- [BOGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computing. In *STOC*, pages 1–10, 1988.
- [CDK91] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI 1991*, pages 318–324, 1991.
- [FBKG02] C. Fernández, R. Béjar, B. Krishnamachari, and C. Gomes. Communication and computation in dis. CSP algorithms. In *CP*, pages 664–679, 2002.
- [FMG02] B. Faltings and S. Macho-Gonzalez. Open constraint satisfaction. In *CP*, 2002.
- [FMW01] E.C. Freuder, M. Minca, and R.J. Wallace. Privacy/efficiency tradeoffs in distributed meeting scheduling by constraint-based agents. In *Proc. IJCAI DCR*, pages 63–72, 2001.
- [Gol04] Oded Goldreich. *Foundations of Cryptography*, volume 2. Cambridge, 2004.
- [GP02] IP. Gent and P. Prosser. An empirical study of the stable marriage problem with ties and incomplete lists. In *ECAI 2002*, pages 141–145, 2002.
- [Ham99] Youssef Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués*. PhD thesis, Université Montpellier II, Juillet 1999.
- [HCN⁺01] T Herlea, J. Claessens, G. Neven, F. Piessens, B. Preneel, and B. Decker. On securely scheduling a meeting. In *Proc. of IFIP SEC*, pages 183–198, 2001.
- [IM02] R. Irving and D. Manlove. The stable roommates with ties. *Journal of Algorithms*, 43(1):85–105, 2002.
- [Kil88] J. Kilian. Founding cryptography on oblivious transfer. In *Proc. of ACM Symposium on Theory of Computing*, pages 20–31, 1988.
- [LV97] Michel Lemaître and Gérard Verfaillie. An incomplete method for solving distributed valued constraint satisfaction problems, 1997.
- [MJ00] P. Meseguer and M. Jiménez. Distributed forward checking. In *CP'2000 Distributed Constraint Satisfaction Workshop*, 2000.

- [MR99] Eric Monfroy and Jean-Hugues Rety. Chaotic iteration for distributed constraint propagation. In *SAC*, pages 19–24, 1999.
- [MTSY04] P.J. Modi, M. Tambe, W.-M. Shen, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *AIJ*, 2004.
- [SGM96] G. Solotorevsky, E. Gudes, and A. Meisels. Algorithms for solving distributed constraint satisfaction problems (DCSPs). In *AIPS96*, 1996.
- [Sha79] A. Shamir. How to share a secret. *Comm. of the ACM*, 22:612–613, 1979.
- [Sil02] Marius-Călin Silaghi. *Asynchronously Solving Distributed Problems with Privacy Requirements*. PhD Thesis 2601, (EPFL), June 27, 2002. <http://www.cs.fit.edu/~msilaghi/teza>.
- [Sil03a] M.C. Silaghi. Arithmetic circuit for the first solution of distributed CSPs with cryptographic multi-party computations. In *IAT*, Halifax, 2003.
- [Sil03b] M.C. Silaghi. Solving a distributed CSP with cryptographic multi-party computations, without revealing constraints and without involving trusted servers. In *IJCAI-DCR*, 2003.
- [Sil04a] M.-C. Silaghi. Secure distributed CSP solvers. <http://www.cs.fit.edu/msilaghi/secure/>, 2004.
- [Sil04b] M.C. Silaghi. Incentive auctions and stable marriages problems solved with $n/2$ -privacy of human preferences. Technical Report CS-2004-11, FIT, 2004.
- [SR04] M. Silaghi and V. Rajeshirke. The effect of policies for selecting the solution of a DisCSP on privacy loss. In *AAMAS*, pages 1396–1397, 2004.
- [SSHF00a] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, pages 917–922, Austin, August 2000.
- [SSHF00b] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with private constraints. In *Proc. of AA2000*, pages 177–178, Barcelona, June 2000.
- [WF02] R.J. Wallace and E.C. Freuder. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss. In *DCR*, pages 176–182, 2002.
- [WS04] R. Wallace and M.C. Silaghi. Using privacy loss to guide decisions in distributed CSP search. In *FLAIRS'04*, 2004.
- [Yao82] A. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, 1982.
- [YDIK98] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, 10(5):673–685, 1998.
- [YSH02a] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *Proc. of the AAMAS-02 DCR Workshop*, Bologna, July 2002.
- [YSH02b] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *CP*, 2002.
- [ZM91] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 1991.
- [ZWW02] W. Zhang, G. Wang, and L. Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *PAS*, 2002.

Comparing Two Implementations of a Complete and Backtrack-Free Interactive Configurator

Sathiamoorthy Subbarayan¹, Rune M. Jensen¹, Tarik Hadzic¹,
Henrik R. Andersen¹, Henrik Hulgaard², and Jesper Møller²

¹ Department of Innovation, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{sathi,rmj,tarik,hra}@itu.dk

² Configit Software A/S,
Vermundsgade 38 B, DK-2100 Copenhagen Ø, Denmark
{jm,henrik}@configit-software.com

Abstract. A product configurator should be complete and backtrack free in the sense that the user can choose freely between any valid configuration and will be prevented from making choices that no valid configuration satisfies. In this paper, we experimentally evaluate a symbolic and search-based implementation of an interactive product configuration algorithm with these properties. Our results show that the symbolic approach often has several orders of magnitude faster response time than the search-based approach due to the precompilation of a symbolic representation of the solution space. Moreover, the difference between the average and worst response time for the symbolic approach is typically within a factor of two, whereas it for the search-based approach may be more than two orders of magnitude.

1 Introduction

Product configuration involves two major tasks: to define a modeling language that makes it easy to specify and maintain a formal product model and to develop a decision support tool that efficiently guides users to desirable product configurations [1]. In this paper we focus on the latter task and investigate a *complete* and *backtrack-free* approach to Interactive Product Configuration (IPC). The input to our IPC algorithm is a product model that consists of a set of product variables with finite domains and a set of product rules. A valid product configuration is an assignment to each variable that satisfies the product rules. The IPC algorithm initially has an empty set of user selected assignments. In each iteration of the algorithm, invalid values in the domain of the variables are pruned away such that each value in the domain of a variable is part of at least one valid configuration satisfying the existing set of assignments. The user then selects his preferred value for any free variable. The algorithm terminates when each variable in the product model is assigned a value. The IPC algorithm is complete in the sense that the user can choose freely between any valid configuration. It is also backtrack-free since the user is prevented from

choosing a variable assignment for which no valid configuration exists. Hence the algorithm never forces the user to go back to a previous configuration state to explore alternative choices.

In this paper we experimentally evaluate a symbolic and search-based implementation of the IPC algorithm. The symbolic implementation is using a two-phase approach [2]. In the first phase, the product rules are compiled into a reduced ordered Binary Decision Diagram (BDD) [3] representing the set of valid configurations (the *solution space*). In the second phase, a fast specialized BDD operation is used to prune the variable domains. The worst-case response time only grows polynomially with the size of the BDD. Thus, the computationally hard part of the configuration problem is fully solved in the offline phase given that the compiled BDD is small. The search-based implementation prunes variable domains by iterating over each possible assignment and searching for a valid configuration satisfying it. This approach can be improved by memorizing all assignments for which no solution exist across iterations and, within each iteration, adding all assignments in a found solution (not just the one being verified).

The symbolic approach has been implemented using *Configit Developer 3.2* [4] that employs a BDD-derived symbolic representation called Virtual Tables (VTs). The search-based approach has been implemented using *ILOG Solver 5.3* [5]. We have developed a publicly available benchmark suite called CLib [6] for the experiments that consists of 14 configuration problems. Our results show that the symbolic approach often has several orders of magnitude faster response time than the search based approach. In addition, the difference between average and worst response time is often within a factor of two for the symbolic approach, but some times more than two orders of magnitude for the search-based approach. Moreover, our results indicate that the configuration space of industrial configuration problems often have small symbolic representations. This result is somewhat surprising since BDDs blow up for many combinatorial problems investigated in AI (e.g., the n -queens problem and other permutation problems). It may be the frequent hierarchical structure of real-world configuration problems that make them particularly well-suited for BDDs.

The remainder of the paper is organized as follows. In Sect. 2, we define product configuration and describe the IPC algorithm. The symbolic and search-based implementation of the IPC algorithm are described in Sect. 3 and Sect. 4, respectively. We focus on describing the symbolic implementation since we assume that the search-based implementation is straight forward for most readers. Section 5 presents experimental work. Finally, we describe related work in Sect. 6 and draw conclusions in Sect. 7.

2 Interactive Product Configuration

We can think of product configuration as a process of specifying a product defined by a set of attributes, where attribute values can be combined only in predefined ways. Our formal definition captures this as a mathematical object

with three elements: variables, domains for the variables defining the combinatorial space of possible assignments and formulas defining which combinations are valid assignments. Each variable represents a product attribute. The variable domain refers to the options available for its attribute and formulas specify the rules that the product must satisfy.

Definition 1. A configuration problem C is a triple (X, D, F) , where X is a set of variables x_1, x_2, \dots, x_n , D is the Cartesian product of their finite domains $D_1 \times D_2 \times \dots \times D_n$, and $F = \{f_1, f_2, \dots, f_m\}$ is a set of propositional formulas over atomic propositions $x_i = v$, where $v \in D_i$, specifying conditions that the variable assignments must satisfy.

Each formula f_i is a propositional expression inductively defined by

$$\phi \equiv x_i = v \mid \phi \wedge \psi \mid \phi \vee \psi \mid \neg\phi, \quad (1)$$

where $v \in D_i$. We use the abbreviation $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$ for logical implication. For a configuration problem C , we define the solution space $S(C)$ as the set of all *valid configurations*, i.e. the set of all assignments to the variables X that satisfy the rules F . Many interesting questions about configuration problems are hard to answer. Just determining whether the solution space is empty is NP-complete, since the Boolean satisfiability problem can be reduced to it in polynomial time.

Example 1. Consider specifying a T-shirt by choosing the color (black, white, red, or blue), the size (small, medium, or large) and the print ("Men In Black" - MIB or "Save The Whales" - STW). There are two rules that we have to observe: if we choose the MIB print then the color black has to be chosen as well, and if we choose the small size then the STW print (including a big picture of a whale) cannot be selected as the large whale does not fit on the small shirt. The configuration problem (X, D, F) of the T-shirt example consists of variables $X = \{x_1, x_2, x_3\}$ representing color, size and print. Variable domains are $D_1 = \{black, white, red, blue\}$, $D_2 = \{small, medium, large\}$, and $D_3 = \{MIB, STW\}$. The two rules translate to $F = \{f_1, f_2\}$, where $f_1 = (x_3 = MIB) \Rightarrow (x_1 = black)$ and $f_2 = (x_3 = STW) \Rightarrow (x_2 \neq small)$. There are $|D_1||D_2||D_3| = 24$ possible assignments. Eleven of these assignments are valid configurations and they form the solution space shown in Fig. 1. \diamond

<i>(black, small, MIB)</i>	<i>(black, large, STW)</i>	<i>(red, large, STW)</i>
<i>(black, medium, MIB)</i>	<i>(white, medium, STW)</i>	<i>(blue, medium, STW)</i>
<i>(black, medium, STW)</i>	<i>(white, large, STW)</i>	<i>(blue, large, STW)</i>
<i>(black, large, MIB)</i>	<i>(red, medium, STW)</i>	

Fig. 1. Solution space for the T-shirt example

By interactive product configuration we refer to the process of a user interactively tailoring a product to his specific needs by using supporting software

called a *configurator*. Every time the user assigns a value to a variable, the configurator restricts the possible solutions to configurations consistent with this new condition. The user keeps selecting variable values until only one configuration is left. The IPC algorithm in Fig. 2 illustrates this interactive process. In line 1, the configurator takes a given configuration problem C and compiles

```

IPC( $C$ )
1    $R \leftarrow \text{COMPILE}(C)$ 
2   while  $|R| > 1$ 
3     do choose  $(x_i = v) \in \text{VALID-ASSIGNMENTS}(R)$ 
4      $R \leftarrow R|_{x_i=v}$ 

```

Fig. 2. The IPC Algorithm

it into an internal representation R . The procedure $\text{VALID-ASSIGNMENTS}(R)$ in line 3 extracts the set of valid assignments (choices) from the internal representation. In line 4, the internal representation is restricted to configurations satisfying the new condition. This behavior of the configurator enforces a very important property of interactive configuration called *completeness of inference*. The user cannot pick a value that is not a part of a valid solution, and furthermore, a user is able to pick all values that are part of at least one valid solution. These two properties are often not satisfied in existing configurators, either exposing the user to backtracking or making some valid choices unavailable. The symbolic and search-based implementation of the IPC algorithm differ in their internal representation and implementation of the valid assignment and restrict operations.

Example 2. For the T-shirt problem, the assignment $x_2 = \textit{small}$ will, by the second rule, imply $x_3 \neq \textit{STW}$ and since there is only one possibility left for variable x_3 , it follows that $x_3 = \textit{MIB}$. The first rule then implies $x_1 = \textit{black}$. Unexpectedly, we have completely specified a T-shirt by just one assignment. \diamond

From the user’s point of view, the configurator responds to the assignment by calculating valid choices for undecided variables. It is important that the response time is very short, offering the user a truly interactive experience. The demand for short response-time and completeness of inference is difficult to satisfy due to the hardness of the configuration problem.

3 Symbolic Implementation of the IPC Algorithm

Since checking whether the solution space is empty is NP-complete, it is unlikely that we can construct a configurator that takes a configuration problem and guarantees a response time that is polynomially bounded with respect to its size. The symbolic approach is offline to compile the configuration problem to

a representation of the solution space that supports fast interaction algorithms. The idea is to remove the hard part of the problem in the offline phase. This will happen if the compiled representation is small. We cannot, however, in general avoid exponentially large representations.

3.1 Symbolic Solution Space Representation

A configuration problem can be efficiently encoded using Boolean variables and Boolean functions. We assume that domains D_i contain successive integers starting from 0. For example we encode $D_2 = \{small, medium, large\}$ as $D_2 = \{0, 1, 2\}$. Let $l_i = \lceil \lg |D_i| \rceil$ denote the number of bits required to encode a value in domain D_i . Every value $v \in D_i$ can be represented in binary as a vector of Boolean values $\mathbf{v} = (v_{l_i-1}, \dots, v_1, v_0) \in \mathbb{B}^{l_i}$. Analogously, every variable x_i can be encoded by a vector of Boolean variables $\mathbf{b} = (b_{l_i-1}, \dots, b_1, b_0)$. Now, the formula $x_i = v$ can be represented as a Boolean function given by the expression $b_{l_i-1} = v_{l_i-1} \wedge \dots \wedge b_1 = v_1 \wedge b_0 = v_0$ (written $\mathbf{b} = \mathbf{v}$). For the T-shirt example we have, $D_2 = \{small, medium, large\}$ and $l_2 = \lceil \lg 3 \rceil = 2$, so we can encode *small* $\in D_2$ as 00 ($b_1 = 0, b_0 = 0$), *medium* as 01 and *large* as 10.

The translation to a Boolean domain is not surjective. There may exist assignments to the Boolean variables yielding invalid values. For example, the combination 11 does not encode a valid value in D_2 . Therefore we introduce a *domain constraint* that forbids these unwanted combinations $F_D = \bigwedge_{i=1}^n (\bigvee_{v \in D_i} x_i = v)$. Furthermore, we define a translation function τ that maps a propositional expression ϕ to the Boolean function it represents

$$\tau(\phi) : \prod_{i=1}^n \mathbb{B}^{l_i} \rightarrow \mathbb{B}. \quad (2)$$

The translation is defined inductively as follows

$$\tau(x_i = v) \equiv (\mathbf{b}_i = \mathbf{v}) \quad (3)$$

$$\tau(\phi \wedge \psi) \equiv \tau(\phi) \wedge \tau(\psi) \quad (4)$$

$$\tau(\phi \vee \psi) \equiv \tau(\phi) \vee \tau(\psi) \quad (5)$$

$$\tau(\neg\phi) \equiv \neg\tau(\phi). \quad (6)$$

Finally we can express a Boolean function representation $S'(C)$ of the solution space $S(C)$.

$$S'(C) \equiv \tau(F_D) \wedge \bigwedge_{j=1}^m \tau(f_j). \quad (7)$$

The resulting symbolic implementation of the IPC algorithm is shown in Fig. 3. In this implementation, the internal representation is a Boolean encoding Sol of the solution space. In order to restrict the solution space in line 4, it is conjoined with the Boolean encoding of the assignment chosen by the user.

```

SYMBOLIC-IPC( $C$ )
1    $Sol \leftarrow S'(C)$ 
2   while  $|Sol| > 1$ 
3     do choose  $(x_i = v) \in \text{VALID-ASSIGNMENTS}(Sol)$ 
4      $Sol \leftarrow Sol \wedge \tau(x_i = v)$ 

```

Fig. 3. Symbolic implementation of the IPC algorithm

3.2 Binary Decision Diagrams

A reduced ordered Binary Decision Diagram (BDD) is a rooted directed acyclic graph representing a Boolean function on a set of linearly ordered Boolean variables. It has one or two terminal nodes labeled 1 or 0 and a set of variable nodes. Each variable node is associated with a Boolean variable and has two outgoing edges *low* and *high*. Given an assignment of the variables, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is *true*, if the label of the reached terminal node is 1; otherwise it is *false*. The graph is ordered such that all paths respect the ordering of the variables.

A BDD is reduced such that no pair of distinct nodes u and v are associated with the same variable and low and high successors (Fig. 4a), and no variable node u has identical low and high successors (Fig. 4b). Due to these reductions,

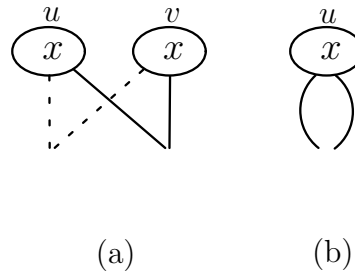


Fig. 4. (a) nodes associated to the same variable with equal low and high successors will be converted to a single node. (b) nodes causing redundant tests on a variable are eliminated. High and low edges are drawn with solid and dashed lines, respectively

the number of nodes in a BDD for many functions encountered in practice is often much smaller than the number of truth assignments of the function. Another advantage is that the reductions make BDDs canonical [3]. Large space savings can be obtained by representing a collection of BDDs in a single multi-rooted

graph where the sub-graphs of the BDDs are shared. Due to the canonicity, two BDDs are identical if and only if they have the same root. Consequently, when using this representation, equivalence checking between two BDDs can be done in constant time. In addition, BDDs are easy to manipulate. Any Boolean operation on two BDDs can be carried out in time proportional to the product of their size. The size of a BDD can depend critically on the variable ordering. To find an optimal ordering is a co-NP-complete problem in itself [3], but a good heuristic for choosing an ordering is to locate dependent variables close to each other in the ordering. For a comprehensive introduction to BDDs and *branching programs* in general, we refer the reader to Bryant’s original paper [3] and the books [7, 8].

3.3 BDD-Based Implementation of the IPC Algorithm

In the offline phase of BDD-based interactive configuration, we compile a BDD $\tilde{S}(C)$ of the Boolean function $S'(C)$ of the solution space. The variable ordering of Boolean variables of $\tilde{S}(C)$ is identical to the ordering of the Boolean variables of $S'(C)$. $\tilde{S}(C)$ can be compiled using a BDD version $\tilde{\tau}$ of the function τ , where each Boolean operation is translated to its corresponding BDD operation

$$\tilde{\tau}(x_i = v) \equiv \text{BDD of } \tau(x_i = v) \quad (8)$$

$$\tilde{\tau}(\phi \wedge \psi) \equiv \text{Op}_{\wedge}(\tilde{\tau}(\phi), \tilde{\tau}(\psi)) \quad (9)$$

$$\tilde{\tau}(\phi \vee \psi) \equiv \text{Op}_{\vee}(\tilde{\tau}(\phi), \tilde{\tau}(\psi)) \quad (10)$$

$$\tilde{\tau}(\neg\phi) \equiv \text{Op}_{\neg}(\tilde{\tau}(\phi)). \quad (11)$$

In the base case (8), $\tilde{\tau}(x_i = v)$ denotes a BDD of the Boolean function $\tau(x_i = v)$ as defined in Sec. 3.1. For each of the inductive cases, we first compile a BDD for each sub-expression and then perform the BDD operation corresponding to the Boolean operation on the sub-expressions. We have

$$\tilde{S}(C) \equiv \text{Op}_{\wedge}(\tilde{\tau}(F_D), \tilde{\tau}(f_1), \dots, \tilde{\tau}(f_m)). \quad (12)$$

Due to the polynomial complexity of BDD operations, the complexity of computing $\tilde{S}(C)$ may be exponential in the size of C .

Example 3. The BDD representing the solution space of the T-shirt example introduced in Sect. 2 is shown in Fig. 5. In the T-shirt example there are three variables: x_1, x_2 and x_3 , whose domain sizes are four, three and two, respectively. As explained in Sect. 3.1, each variable is represented by a vector of Boolean variables. In the figure the Boolean vector for the variable x_i with domain D_i is $(x_i^{l_i-1}, \dots, x_i^1, x_i^0)$, where $l_i = \lceil \lg |D_i| \rceil$. For example, in the figure, variable x_2 which corresponds to the size of the T-shirt is represented by the Boolean vector (x_2^1, x_2^0) . In the BDD any path from the root node to the terminal node 1, corresponds to one or more valid configurations. For example, the path from the root node to the terminal node 1, with all the variables taking low values represents the valid configuration (*black, small, MIB*). Another path with x_1^1, x_1^0 ,

and x_2^1 taking low values, and x_2^0 taking high value represents two valid configurations: $(black, medium, MIB)$ and $(black, medium, STW)$, namely. In this path the variable x_3^0 is a don't care variable and hence can take both low and high value, which leads to two valid configurations. Any path from the root node to the terminal node 0 corresponds to invalid configurations. \diamond

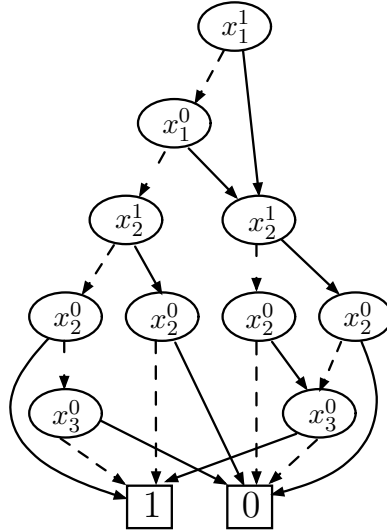


Fig. 5. BDD of the solution space of the T-shirt example. Variable x_i^j denotes bit b_j of the Boolean encoding of product variable x_i .

For a BDD version of the SYMBOLIC-IPC algorithm, each Boolean operation is translated to its corresponding BDD operation. The response time is determined by the complexity of performing a single iteration of the procedure. All sub-operations can be done in time linear in the size of Sol except VALID-ASSIGNMENTS in Line 3. This procedure can be realized by a specialized BDD operation with worst-case complexity

$$O\left(\sum_{i=1}^n |V_i| |D_i|\right), \quad (13)$$

where V_i denotes the nodes in Sol associated with BDD variables encoding the domain of variable x_i . As usual, D_i denotes the domain of x_i . For each value of each variable, the procedure tracks whether the value is encoded by Sol . Due to the ordering of the BDD variables, for each variable x_i , this tracking can be constrained to the nodes V_i .

4 Search-based Implementation of the IPC Algorithm

Search-based approaches like SAT techniques or CSP techniques can also be used to implement the IPC algorithm. In such cases the first step of the algorithm is to compile the configuration problem into an internal representation used by the applied search technique. The internal representation implicitly represents the set of valid configurations. The iterative step of the IPC algorithm is repeated until all the variables in the product model are assigned a value. For search-based techniques, the procedure for calculating VALID-ASSIGNMENTS is theoretically intractable. Every time an assignment is made, propagation is applied to prune the invalid values in the domain of all unselected variables. This can be done by checking whether each value in the domain of all the variables has at least one valid configuration satisfying the existing set of variable assignments. In the worst case, the search-based configurator has to find $1 - m + \sum_{i=1}^m d_i$ solutions for each iteration, where m is the number of unassigned variables during the iteration, and d_i is the domain size of the unassigned variable i during the iteration. Improved average complexity of VALID-ASSIGNMENTS can be obtained by memorizing invalid variable values. Since the solution space keeps decreasing, an invalid value will remain invalid in later iterations. In addition, within each iteration, whenever the search-based IPC finds a solution, it can mark all the assignments in the solution as valid. This can remove redundant search for solutions.

5 Experimental Evaluation

We use *Configit Developer 3.2* [4] for the symbolic implementation of the IPC algorithm. Configit Developer has a compiler which first checks for the semantic correctness of the product model. If the semantics is valid, it creates a Virtual Table (VT) that symbolically represents all valid solutions of the product model. For the search-based implementation of the IPC algorithm we use *ILOG Solver 5.3* [5]. ILOG Solver is a commercial constraint programming based C++ library for solving constraint satisfaction/optimization problems. In a comparative study, it has been shown to outperform a range of other CSP solvers [9]. The source code of our benchmarks for ILOG Solver has been constructed to the best of our knowledge. But we are not experts on the ILOG Solver technology and there may exist more efficient implementations. ILOG Configurator [5] is another product from the same company which uses ILOG Solver for configuration problems. ILOG Configurator, however, only does set bound propagation. This means that it may be possible for a user to choose assignments for which no valid configuration exists if the ILOG Configurator is used for interactive configuration.

For each configuration benchmark used in the experiments, 1000 random requests are generated using Configit Developer. There are several cycles of requests. Each cycle contains a sequence of requests in increasing order. Each request consists of a set of assignments to some of the variables in the product

model. Each cycle of requests simulates an interactive configuration process of a product.

All the experiments were carried out on a Pentium-III 1 GHz machine with 2GB of memory, running Linux. The benchmarks used in the experiments are available in the CLib benchmark suite [6] in the Configit input language and the ILOG Solver source language.

The results are shown in Table 1. The first column in the table lists the name of the benchmark. The second column lists the amount of CPU time in seconds used by Configit Developer for generating the corresponding VT file. The third column lists the size of the generated VT file in Kilo Bytes (KB). As for BDDs (see Sect. 3), the variable ordering plays a crucial role in the size of the VT files generated. Unless specified, the default variable order was used in the experiments. The fourth column lists the number of solutions (#Sol) in the generated VT file. After generating a VT, the number of solutions represented by it can easily be counted. This is equivalent to the total number of valid configurations for the instance. In the subsequent four columns, the CPU time for Average response (Avg. RT) and Worst response (Wst. RT) are listed in seconds for both Configit Developer and ILOG Solver. The response times listed for Configit Developer include the time taken for the requests generation and writing the requests into a file. The corresponding times listed for ILOG Solver only include the time taken for reading the requests information from the file. Requests are generated as specified above.

Table 1. Experimental Results

Benchmark	Virtual Table			Avg. RT (sec)		Wst. RT (sec)	
	Time(sec)	Size(KB)	#Sol	Configit	ILOG	Configit	ILOG
Renault	460.00	1292	2.8×10^{12}	0.1273	489.29*	0.240	489.29*
Bike	0.45	22	1.3×10^8	0.0005	1.855	0.010	882.68
PC	0.89	24	1.1×10^6	0.0007	1.302	0.010	2.12
PSR	0.38	37	7.7×10^9	0.0014	2.398	0.010	486.12
Parity32_13	30.00	1219	2.0×10^8	0.0960	0.061	0.416	0.24
Big-PC	14.82	76	6.2×10^{19}	0.0012		0.010	
v1	5.67 [†]	253	8.2×10^{123}	0.1620		0.320	
w1	56.52 [†]	1347	1.0×10^{89}	0.0680		0.160	
ESVS	0.25	6.7	3.5×10^9	0.0004	0.059	0.010	0.14
FS	0.25	5.8	2.4×10^7	0.0003	0.036	0.010	0.21
FX	0.22	5.3	1.2×10^6	0.0003	0.029	0.010	0.10
Machine	0.14	6.7	4.7×10^8	0.0004	0.009	0.010	0.03
C169_FV	2.30 (144)	287	3.2×10^{15}	0.0134	0.195	0.010	28.77
C211_FS	6.93 (957)	370	1.4×10^{67}	0.0219	0.314	0.020	67.09
C250_FV	3.22 (111)	308	1.2×10^{37}	0.0148	0.203	0.010	38.98
C638_FVK	16.53 (1980)	534	8.8×10^{123}	0.0385	0.608	0.050	72.62

*For finding one solution only (i.e., not complete).

[†]The variable order file has been provided by Configit Software.

The Renault car configuration benchmark is described in [10]. Configit Developer takes 460 seconds to generate the VT file containing 2.8×10^{12} solutions for the Renault instance. But ILOG Solver takes 489 seconds just to solve the problem represented as a CSP instance. The average response time obtained for Renault by Configit Developer is 0.1273 second. Corresponding worst response time is 0.240 second.

The Bike and PC instances are configuration examples provided along with Configit Developer. They represent a bike and a personal computer configuration problem. The size of the PC and Bike instances in the Configit language is around 500 and 700 lines of code, respectively. In both cases, the average response time for Configit Developer is only a fraction of a millisecond. This is possible as the sizes of the corresponding VT files are very small. The corresponding worst response time for those two instances are 10 milliseconds only. But the average and worst response times for ILOG Solver are comparatively very high. The worst response time for ILOG Solver in case of Bike is 882.68 seconds, which is more than 400 times the corresponding average response time. In case of the PC example, there is not a large difference between the average and the worst response time of ILOG Solver. But still the values are high when compared to the corresponding values for Configit Developer. The PSR benchmark represents a power supply restoration problem modelled as a configuration instance. Further details about this problem is available in [11]. The performance of ILOG Solver and Configit Developer on the PSR instance is similar to those obtained for the Bike instance, with a large worst case response time by ILOG Solver. Parity32_13 represents a parity learning problem as a configuration instance. Information about this problem is available in [12]. In case of the Parity32_13 problem, ILOG Solver has better performance compared to Configit Developer. The difference is not large though. It is interesting that the VT files for Bike, PC, and PSR are compiled faster than the corresponding average response times given by ILOG Solver.

Big-PC represents the configuration problem of a personal computer with several additional features than those of the previous PC instance. The v1 and w1 instances represent real configuration problems of some of the customers of Configit Software. They are anonymized by renaming.³ For these two instances, the corresponding variable order file was also provided by Configit Software. These instances have very large product models. For example, the Big-PC has 2500 lines of code in the Configit language. The w1 instance has more than 66,000 lines of code. It is very hard and error prone to convert them manually to ILOG Solver code. Hence for those instances Table 1 only has results obtained for Configit Developer. Even though these instances represent very large product models, Configit Developer has fast average and worst response times.

The ESVS, FS, FX, and Machine instances are from [13]. The first three instances represent screw compressor configuration problems. The last one represents parts of a real-world 4-wheeled vehicle configuration problem, anonymized by renaming. These four instances are easy compared to the previous 8 instances.

³ Due to legal issues, v1 and w1 are not available in CLib.

This is reflected by their VT file sizes. The VT files for these instances are generated in a fraction of a second. In case of the FS instance, the VT file generation time is almost equal to the corresponding worst response time by ILOG Solver.

The last four instances are automotive product configuration problems from [14]. The original instances are in DIMACS CNF format. Good variable orders for these instances are obtained with the BuDDy BDD package [15] using the sift dynamic variable ordering heuristic. Configit Developer uses these variable orders for efficient VT file generation. The time taken by BuDDy to find a good variable order is listed in brackets in the second column of the table. For some large CNF files in [14], we did not do experiments as it took a lot of time to find good variable orders. For those instances, the average response time and the worst response time given by ILOG Solver are very large.

6 Related Work

Compilation techniques have also been studied for search techniques. In [16], the authors presented Minimal Synthesis Trees (MSTs), a data structure to compactly represent the set of all solutions in a CSP. It takes advantage of combining the consistency techniques with a decomposition and interchangeability idea. The MST is a polynomial-size structure. Operations on the MSTs, however, are of exponential time complexity. This may lead to long response times for an interactive configurator based on MSTs.

Acyclic constraint networks and the tree clustering algorithm [17, 18] represent a CSP solution space in a more compact way, organizing it as a tree of solved sub-problems. For extracting a solution, the generated structure offers polynomial time guarantees in the size of the structure. The size of the sub-problems, however, cannot be controlled for all instances and might lead to an exponential blow-up. The complexity of the original problem is dominated by the complexity of the sub-problems, which are exponential in both space and time. Nevertheless, this is one of the first compilation approaches used to solve CSP problems. There are efforts to cope with this exponential blow-up by additional compression using Cartesian product representation [19].

We are only aware of one other symbolic precompilation technique. In [10], the authors present a method which compiles all valid solutions of a configuration problem into an automaton. After compiling the solutions into an automaton, functions required for interactive configuration, like implications, explanations, and valid-domain-calculations can be done efficiently. They also present a theoretical view of all the complexity issues involved in their approach. They show that all the tasks involved in an interactive configuration process are intractable in the worst case. The BDD and automata approach to two-phase interactive configuration may perform equally well. A major advantage of using BDDs, however, is that this data structure has been studied intensely in formal verification for representing formal models of large systems [20, 21]. In particular, the variable ordering problem is well studied [7]. Furthermore a range of powerful

software packages have been developed for manipulating BDDs [15, 22]. To our knowledge, automata compilation has not reached this level of maturity.

Previous work comparing symbolic and search-based approaches is very limited. Comparative studies of the SAT search engine zChaff [23] and BDD compilation show that neither approach has generally better performance than the other [24]. Similarly, a comparison of the Davis-Putnam procedure and BDDs shows complementary strengths of the two approaches rather than one dominating the other [25]. For interactive configuration, though, an important advantage of the BDD approach is that it is possible to compile the solution space prior to user interaction whereas search must be interleaved with user requests.

7 Conclusion

In this paper we have compared a symbolic and search-based implementation of a complete and backtrack-free interactive product configuration algorithm. Our experimental results show that the symbolic approach often has several orders of magnitude faster response time than the search based approach due to the precompilation of the solution space into a symbolic representation. In addition, the difference between average and worst response time is often much smaller for the symbolic approach than for the search-based approach. Our results indicate that BDD-derived representations often are small for real-world configuration instances. We believe that this may be due to the modular structure of configuration problems with a frequent hierarchical tree-like decomposition of dependencies that BDDs are particularly well-suited for.

We are currently working on an open source C++ Library for BDD-based interactive configuration for research and education purposes (CLab, [26]) and a comprehensive benchmark suite of industrial configuration problems (CLib, [6]). Future work includes developing specialized BDD operations to support both the compilation phase and interactive phase of product configuration.

Acknowledgments

We would like to thank ILOG for their elaborate answers to our user questions for ILOG Solver 5.3. We also thank Erik R. van der Meer for providing the T-shirt example.

References

1. Sabin, D., Weigel, R.: Product configuration frameworks - a survey. *Intelligent Systems, IEEE* **13** (1998) 42–49
2. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: *Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems, DTU-tryk* (2004) 131–138

3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* **8** (1986) 677–691
4. Configit Software A/S. <http://www.configit-software.com> (online)
5. ILOG. <http://www.ilog.com> (online)
6. CLib: Configuration benchmarks library. <http://www.itu.dk/doi/VeCoS/clib/> (online)
7. Meinel, C., Theobald, T.: *Algorithms and Data Structures in VLSI Design*. Springer (1998)
8. Wegener, I.: *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics (SIAM) (2000)
9. Fernández, A.J., Hill, P.M.: A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints* **5** (2000) 275–301
10. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence* **1-2** (2002) 199–234 <ftp://fpt.irit.fr/pub/IRIT/RPDM/Configuration/>.
11. Thiébaux, S., Cordier, M.O.: Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In: *Pre-Proceedings of the 6th European Conference on Planning (ECP-01)*. (2001) 85–96
12. Parity-Function. <ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/crawford/README> (online)
13. Tiihonen, J., Soinen, T., Niemelä, I., Sulonen, R.: Empirical testing of a weight constraint rule based configurator. In: *ECAI 2002 Configuration Workshop*. (2002) 17–22 <http://www.soberit.hut.fi/pdmg/Empirical/index.html>.
14. Sinz, C., Kaiser, A., Kchlin, W.: Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **17** (2003) 75–97 Special issue on configuration <http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/>.
15. Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. <http://sourceforge.net/projects/buddy> (online)
16. Weigel, R., Faltings, B.: Compiling constraint satisfaction problems. *Artificial Intelligence* **115** (1999) 257–287
17. Dechter, R., Pearl, J.: Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence* **34** (1987) 1–38
18. Dechter, R., Pearl, J.: Tree-clustering schemes for constraint-processing. *Artificial Intelligence* **38** (1989) 353–366
19. Madsen, J.N.: *Methods for interactive constraint satisfaction*. Master’s thesis, Department of Computer Science, University of Copenhagen (2003)
20. Burch, J.R., Clarke, E.M., McMillan, K.: Symbolic model checking: 10^{20} states and beyond. In: *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*. (1990) 428–439
21. Yang, B., Bryant, R.E., O’Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of BDD-based model checking. In: *Formal Methods in Computer-Aided Design FMCAD’98*. (1998) 255–289
22. Somenzi, F.: CUDD: Colorado university decision diagram package. <ftp://vlsi.colorado.edu/pub/> (1996)
23. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: *Proceedings of the 38th Design Automation Conference (DAC’01)*. (2001)
24. Pan, G. and Vardi, M.Y.: Search vs. symbolic techniques in satisfiability solving. In: *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*. (2004)

25. Uribe, T.E., Stickel, M.E.: Ordered binary decision diagrams and the Davis-Putnam procedure. In Jouannaud, J.P., ed.: Proceedings of the 1st International Conference on Constraints in Computational Logics. Volume 845 of Lecture Notes in Computer Science., Springer (1994)
26. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. <http://www.itu.dk/people/rmj/clab/> (online)

BDD-based Recursive and Conditional Modular Interactive Product Configuration

Erik R. van der Meer and Henrik Reif Andersen

Department of Innovation, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{ervandermeer,hra}@itu.dk

Abstract. Interactive product configuration is a difficult problem in constraint programming. One of the reasons for this is that interactivity requires that the system always responds fast. One of the ways to deal with this requirement is to adopt a two-stage approach in which a product model is compiled off-line before user interaction takes place. One such approach uses reduced ordered Binary Decision Diagrams (BDDs) as the compilation target for the product model. Advantages of this representation are that it is often quite compact, that interactive configuration can take place in time linear of the size of the representation, and that this size is only dependent on the solution set (the semantics) and not on the description (the syntax) of the product model. In this paper, we extend this approach with the notion of modules, which are more loosely coupled parts of a configuration problem that often correspond with actual modules in the actual product. We allow for the combining of such modules by means of shared variables, for conditional instantiation of such modules, and for recursion between modules which leads to more natural descriptions of modular systems that could, in principle, be a priori unbounded in size.

1 Introduction

Interactive product configuration is often presented as an application in constraint programming [1, 2]. One of the main problems in implementing a good interactive product configurator is the requirement of responding sufficiently fast although the problem is NP-hard. Therefore, sometimes an off-line pre-computation phase is used to crack the problem so that the interactive phase can work in time polynomial in the size of the intermediate representation [3, 4]. One such approach is based on compilation to reduced ordered Binary Decision Diagrams (BDDs), which can represent many practical solution sets very compactly.

This is the approach we have chosen to extend. In this paper, we present the formal semantics of the pre-computation and interaction phases of the recursive and conditional modular interactive configuration problem. In a modular configuration problem, each module is a configuration problem by itself, but the modules can share variables, as long as the dependency graph for the modules

is a directed acyclic graph, and the run-time dependency graph for the module instances is a tree. In the conditional modular configuration problem, module instances in the tree exist or not depending on conditions in the parent module instance. In the recursive conditional modular configuration problem, the dependency graph for the modules is a directed graph, and the run-time dependency graph for the module instances is a tree of unbounded but finite size.

We achieve strong guarantees on user guidance, in that all configuration is backtrack-free, while the system is sound (no invalid/inconsistent solutions reachable), complete (all finite valid consistent solutions reachable), and terminable (finite valid consistent solution reachable at all times). This can be achieved in run-time linear in the number of modules. However, where in the non-modular configuration problem the run-time can be exponential in the size of the original problem, the modular configuration problem can take time exponential in the sizes of the original modules, or time exponential in the combined domain sizes of the shared variables. (If it did not, $P = NP$ could be answered affirmatively.) However, typical configuration is very efficient.

We have implemented the configuration system described above, and are currently using it to model a real world modular configuration problem. Thus far, we have studied a 2600 line model consisting of 28 modules, which compiles within a second on a 500 MHz Pentium III, and on which user interaction is handled without any noticeable delay. The model is not finished yet, but the BDDs seem to be growing gently as information is added. Thus far, the largest BDD found (an intermediate result during compilation) is smaller than 2000 nodes. Given the strong guarantees our system provides, we believe that we are achieving good results.

2 BDD-based Modular Interactive Configuration

In the modular configuration problem, a module is an entity with variables, domains, and constraints of its own. It can be instantiated by another module, with which it then shares the variables it exports, and it can instantiate other modules, with which it then shares the variables they export. In this way, the module's instances form a tree at run-time.

In programming language terms, a module can to some extent be thought of as a class, from which objects are instantiated. Also, the usual hierarchy building is possible. A car can be a module, with submodules like door, roof, and engine, and each of those can in turn have submodules.

2.1 Language

Our modular configuration language has these statements:

```

module M;
export x1, ..., xn;
import M as I if C;
define x : v1, ..., vn;
ensure C;

```

where x is a variable name, v a value name, M a module name, I an instance name, and C a constraint satisfying the following grammar:

$$\begin{array}{l}
C ::= (C) \\
\quad | \quad !C \\
\quad | \quad C_1 \text{ op } C_2 \text{ where } op \in \{ \&, |, \rightarrow, \leftrightarrow \} \\
\quad | \quad x \text{ op } v \text{ where } op \in \{ <, =, >, <=, <>, >= \} \\
\quad | \quad x_1 \text{ op } x_2 \text{ where } op \in \{ <, =, >, <=, <>, >= \}
\end{array}$$

Names are strings of alphanumerical characters, underscores and periods. The period is only used in names that refer to variables that are shared with imported modules.

An example of a single module: Here is a small t-shirt example, which consists of a single module:

```

module t_shirt;

define size : small, medium, large;
define color : red, blue, black, white;
define print : save_the_whale, men_in_black;

ensure print = save_the_whale -> size > small;
ensure print = men_in_black -> color = black;

```

The semantics of this will be intuitively clear. However, it is interesting to note that even though the example is trivially simple, not many people realize that when they want a small t-shirt, everything is decided.

An example of recursion: Next, we show an example that shows a toy application for recursion; that of configuring a USB tree:

```

module USB;

define type : unused, scanner, printer, camera, hub;

import scanner as scanner if type = scanner;
import printer as printer if type = printer;

```

```

import camera as camera if type = camera;

import USB as hub_connection_1 if type = hub;
import USB as hub_connection_2 if type = hub;
import USB as hub_connection_3 if type = hub;
import USB as hub_connection_4 if type = hub;

```

This shows the power of being able to use recursion in combination with conditional instantiation.

An example of potential conflict: Next, we show a short example showing a potential problem with the conditionality of import. Here is module A:

```

module A;

define x : 0, 1, 2;

import B as b if x = 1 | x = 2;

ensure x = 1 -> b.x <= b.y;
ensure x = 2 -> b.x < b.y;

```

And here is module B:

```

module B;

export x, y;
define x : 0, 1;
define y : 0, 1;

ensure x >= y;

```

The point here is that, while selection of $x = 1$ is fine, $x = 2$ leads to a conflict. Therefore, 2 should be removed from the domain of variable x . Please note that, since instantiation conditions can be arbitrarily large expressions, and since conflicts can arise from any combination of modules, this is not necessarily easy. However, our configurator handles this correctly while maintaining fast run-time response, stopping the user if and only if necessary.

An example of potential infinity: Next, we show a short example showing a potential problem with the recursiveness of import. Here is module A:

```

module A;

```

```

export x;
define x : 0, 1, 2;

import A if x = 1 | x = 2;

ensure x = 2 -> A.x = 2;

```

The point here is that, while selection of $x = 1$ is fine, $x = 2$ leads to the requirement to instantiate an infinite number of modules. Clearly, this needs to be avoided, again by removing 2 from the domain of x . Again, this situation can occur in more complicated ways, for example when different modules instantiate each other recursively. In fact, the same module can even occur multiple times in such a loop! However, our configurator handles this correctly while maintaining fast run-time response, again only stopping the user if and only if necessary.

Another example of potential infinity: Next, we show a short example showing a related potential problem with the recursiveness of import. Here is module A:

```

module A;

export x;
define x : 0, 1, 2, 3, 4;

import A as a1 if x = 1 | x = 3;
import A as a2 if x = 2 | x = 4;

ensure x = 1 -> a1.x <= 2;
ensure x = 2 -> a2.x <= 2;
ensure x = 3 -> a1.x >= 3;
ensure x = 4 -> a2.x >= 3;

```

The point here is that, while selection of $x = 0, 1$ or 2 is fine, $x = 3$ or 4 leads to trouble. The problem here is again one of instantiating an infinite number of modules, only now in lock step with user choices. That is, the configurator does not have to instantiate an infinite number of modules right away, but the user is doomed to keep on configuring for ever and ever. A configurator should avoid this, by removing the values 3 and 4 from the domain of x . Again, this problem can occur in more complicated forms. Still, our configurator handles this correctly while maintaining fast run-time response, stopping the user if and only if necessary.

2.2 Main Concepts of the Semantics

Compiling a single-module configuration problem is the process of determining the solution set of that module, that is the set of all combinations of values for the variables that satisfy the constraints. The user interaction process of single-module configuration is the process of beginning with that solution space, and reducing it step by step as the user assigns values to variables. During this process, we constantly keep the user informed about the remaining values that she can still choose for the still unassigned variables.

In modular configuration, this picture has to be extended. During modular configuration, there will be a tree of module instances, each of which has a solution space that we call the *local solution space*. Each of these will be stepwise reduced by the user, until their final configurations are found. Of course, it is necessary for the configurator to maintain global consistency over these local solution spaces. That is to say that each solution in a local solution space should be consistent with at least one solution in each of the neighbouring module instances' solution spaces. Consistent here means that the same variables (say x in child B, and $b.x$ in parent A) have the same values.

The *global solution space* is the product of all the local solution spaces. Unlike the local solution spaces, the global solution space can also be made larger by user selections. This happens when a user selection leads to the creation of a new instance, which introduces a new local solution space.

An instance is created when an instantiation condition in an import statement somewhere becomes true for all remaining solutions in the local solution space. In that case, it is known that for all configurations that can come out of the configuration process, the instance is going to be part of the global configuration.

There are three main things being done in the semantics. The first has to do with the propagation of variable names and domains, the second with the prevention of conflicts because of instantiation, and the third with the prevention of infinity problems.

Let us first consider the propagation of variable names and domains. Basically, when a variable is exported in a module, and another module imports that module, then the variable name and its domain are automatically added to the importing module. For example, if module B exports its variable x , then module A which imports module B (with instance name B1, say) automatically gets a variable B1.x. Since modules can import each other recursively, and since it is also possible for a module to export a variable it imported (say A exports B1.x), this propagation is formulated as a fixpoint operation.

Next, conflicts because of instantiation are avoided during the compilation phase: Whenever a module A imports a module B, the constraints that module B imposes on module A when it gets instantiated are already imposed implicitly on module A at compile time. This ensures that any local solutions in A that would instantiate B, but are not compatible with it, are removed from A's solution set.

Finally, the possibility of infinite regression at run-time is avoided at compile-time by a second fixpoint iteration. After the modules have been compiled sepa-

rately (to determine their local solution spaces), the compiler starts this fixpoint iteration to determine which of the local solutions of a module would allow that module to be at the root of a finite subtree in the instance tree. Only those solutions will make it to the final solution set. The fixpoint iteration starts with the local solutions for modules that do not instantiate any submodules, and then adds repeatedly those local solutions for modules that only instantiate submodules that have compatible solutions that have already been discovered in this fixpoint iteration.

It is important to note that our prototype configuration engine does not represent sets by explicit enumeration of their elements, but that it uses BDDs instead. This makes it a lot more efficient in practice. Furthermore, during the interaction phase, no changes are made in the BDDs, unless the user changes focus from one module to the other, and even then changes are only made to BDDs belonging to the modules along the path between these two. This is not readily apparent from the semantics, though.

2.3 Basic Definitions for the semantics

We now give the semantics of the modular configuration problem. These semantics only deal with user selections, not deselections, and they only contain equivalence as a comparison operator. We do not consider these to be serious drawbacks, because the meaning of deselection can be given in terms of a replay of the remaining selections, and the other comparison operators can easily be translated into a disjunctive subformula. The semantics are presented in this way in order to keep them simpler. However, our prototype engine implements everything directly.

We start the description of the semantics of a modular configuration problem with some preliminary definitions. First, we define \mathcal{M} , which is a sequence of modules.

$$\mathcal{M} = \langle M_1, \dots, M_n \rangle$$

Then there is the *addprefix* function, which takes an instance name and a sequence of variable names, and returns the sequence of variable names with the instance name (and a dot) prefixed to each of them.

$$\text{addprefix}(\text{instname}, \langle x_1, \dots, x_n \rangle) = \langle x'_1, \dots, x'_n \rangle$$

where

$$x'_i = \text{instname} \cdot \langle \cdot \rangle \cdot x_i$$

More important is the notion of projecting a tuple of values (representing a configuration of a single module) onto the interface with another module.

The *project* function here will take a tuple in which the values are from the domains of the variables in the sequence X , and will transform it into a tuple in which the values are from the domains of the variables in the sequence X' . All

variables in X' must occur in X , but the reverse does not have to be the case. Note the trick with the function i , which denotes the mapping from X to X' . It is defined in the second subequation, and used in the right hand side of the main equation.

$$\begin{aligned} \text{project}(X, X', (v_1, \dots, v_n)) &= (v_{i_1}, \dots, v_{i_m}) \\ \text{where} \\ \langle x_1, \dots, x_n \rangle &= X \\ \langle x_{i_1}, \dots, x_{i_m} \rangle &= X' \end{aligned}$$

Finally, there are the notions of projecting a set of tuples onto a set of shorter tuples, and of embedding a set of shorter tuples into a set of tuples. These operations are used to project solution sets in one module onto the interface to another, and then to embed them into the other solution domain.

Please note that `projectset` *projects* from X to X' , whereas `embedset` *embeds* into X from X' , and that X' should be the shorter sequence of variables of the two. In the case of `embedset`, we also need to pass in the entire domain D (the set of all valid and invalid configurations), since we need to know the full domains of the variables in X that do not occur in X' . S denotes a set of solutions, that is, a set of valid configurations within a module, and S' denotes such a set projected onto an interface.

$$\begin{aligned} \text{projectset}(X, X', S) &= \{\text{project}(X, X', V) \mid V \in S\} \\ \text{embedset}(X, X', D, S') &= \{V \mid V \in D, \text{project}(X, X', V) \in S'\} \end{aligned}$$

2.4 Compilation of Names

The first phase in the compilation of a modular configuration problem consists of the propagation of the definitions of exported variables from children to parents. Since modules can import each other recursively, this propagation is carried out as a fixpoint iteration.

The following functions obtain sequences of variable names (X), exported variable names (X^E), and the corresponding domains (D) from (sequences of) `define` and `export` statements. Note that the domain D is not a sequence of the domains of the separate variables, but the cartesian product of those domains. That is, it is a set of tuples, each of which represents a configuration (which may or may not be valid.)

$$\begin{aligned} \llbracket \text{define } x:v_1, \dots, v_n \rrbracket &= (\langle x \rangle, \{v_1, \dots, v_n\}) \\ \llbracket \text{defn}_1; \dots; \text{defn}_n \rrbracket &= (X_1 \cdot \dots \cdot X_n, D_1 \times \dots \times D_n) \\ \text{where} \\ (X_i, D_i) &= \llbracket \text{defn}_i \rrbracket \end{aligned}$$

$$\llbracket \mathbf{export} \ x_1, \dots, x_n \rrbracket = \langle x_1, \dots, x_n \rangle$$

$$\llbracket \mathit{expt}_1; \dots; \mathit{expt}_n \rrbracket = (X_1^E \cdot \dots \cdot X_n^E)$$

where

$$X_i^E = \llbracket \mathit{expt}_i \rrbracket$$

The next function obtains the sequence of variables and the corresponding domain that are added to a module because of the exported variables in the imported module. The function \mathcal{N} is used to look up the information on the imported module (N), which is a tuple, and we use the notation X_N and D_N to select the X and D fields from that tuple.

$$\llbracket \mathbf{import} \ \mathit{modname} \ \mathbf{as} \ \mathit{instname} \ \mathbf{if} \ \mathit{cond} \rrbracket_{\text{XD}} \mathcal{N} = (X, D)$$

where

$$N = \mathcal{N}(\mathit{modname})$$

$$X = \mathit{addprefix}(\mathit{instname}, X_N^E)$$

$$D = D_N^E$$

$$\llbracket \mathit{impt}_1; \dots; \mathit{impt}_n \rrbracket_{\text{XD}} \mathcal{N} = (X_1 \cdot \dots \cdot X_n, D_1 \times \dots \times D_n)$$

where

$$(X_i, D_i) = \llbracket \mathit{impt}_i \rrbracket_{\text{XD}} \mathcal{N}$$

Then, we define the meaning of a module (as far as names are concerned) by the following function, which takes the contributions to the variables and domain from the different sources (local definitions, and imported definitions), and puts that information into a 4-tuple. Note that this semantic function for a module can be seen as returning a mapping from the function \mathcal{N} (which maps module names to these 4-tuples) to a pair mapping the present module name to such a 4-tuple.

To examine the function more closely, the first two subequations define the local and imported contributions to the sequence of variable names and to the domain. The next two subequations define the composition of this information¹. The following two define what part of this information pertains to variables that are exported. This will be taken into account in interpreting the importing of these modules in the next fixpoint iteration. The last line, finally, constructs the 4-tuple N .

¹ The operators here are to be understood as simply composing their operands flatly, that is, not to introduce hierarchical structure.

$\llbracket \mathbf{module} \text{ } modname; expts; impts; defns; ensns \rrbracket_{XD} \mathcal{N} = (modname, N)$

where

$$\begin{aligned} (X_L, D_L) &= \llbracket defns \rrbracket \\ (X_I, D_I) &= \llbracket impts \rrbracket_{XD} \mathcal{N} \\ X &= X_L \cdot X_I \\ D &= D_L \times D_I \\ X^E &= \llbracket expts \rrbracket \\ D^E &= projectset(X, X^E, D) \\ N &= (X, D, X^E, D^E) \end{aligned}$$

$\llbracket modl_1; \dots; modl_n \rrbracket_{XD} \mathcal{N} = \{P_1, \dots, P_n\}$

where

$$P_i = \llbracket modl_i \rrbracket_{XD} \mathcal{N}$$

Finally, the meaning of a modular configuration problem posed as a sequence of modules, is given as a function from the module names to the 4-tuples containing the information pertaining to variable sequences, the domains of these modules, and the parts of those that are exported. This function is given by:

$$\mathcal{N} = \mu F_N. \llbracket \mathcal{M} \rrbracket_{XD} F_N$$

2.5 Compilation of Solutions

The second phase in the compilation of a modular configuration problem consists of the computation of the solution sets of all modules, and the conditions for the instantiations of those modules. In order to maintain global consistency, and since modules can import each other recursively, this needs to be computed by a fixpoint iteration.

We begin with the functions that obtain the subset of configurations that satisfy constraints (S), given the sequence of variable names and the domain:

$\llbracket \mathbf{ensure} \ e \rrbracket XD = \llbracket e \rrbracket$

where

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= X \\ \llbracket x = v \rrbracket &= \{(v_1, \dots, v_n) \in D \mid x_i = x, v_i = v\} \\ \llbracket e_1 \wedge e_2 \rrbracket &= \llbracket e_1 \rrbracket \cap \llbracket e_2 \rrbracket \\ \llbracket e_1 \vee e_2 \rrbracket &= \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket \\ \llbracket \neg e \rrbracket &= D \setminus \llbracket e \rrbracket \end{aligned}$$

$\llbracket ensn_1; \dots; ensn_n \rrbracket XD = (S_1 \cap \dots \cap S_n)$

where

$$S_i = \llbracket ensn_i \rrbracket XD$$

Next, we have the functions that obtain the constraints coming from submodules (S) and the instantiation conditions for those submodules (Δ). Note the equation that defines S , which reflects the fact that constraints from submodules are taken into account if and only if that submodule exists. (That is, when the final configuration is known to be an element of C .) Also, note the function \mathcal{A} which we use to look up the information about the imported module.

$\llbracket \text{import } \textit{modname} \text{ as } \textit{instname} \text{ if } \textit{cond} \rrbracket_{S\Delta} XDA = (S, \Delta)$

where

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= X \\ \llbracket x = v \rrbracket &= \{(v_1, \dots, v_n) \in D \mid x_i = x, v_i = v\} \\ \llbracket c_1 \wedge c_2 \rrbracket &= \llbracket c_1 \rrbracket \cap \llbracket c_2 \rrbracket \\ \llbracket c_1 \vee c_2 \rrbracket &= \llbracket c_1 \rrbracket \cup \llbracket c_2 \rrbracket \\ \llbracket \neg c \rrbracket &= D \setminus \llbracket c \rrbracket \\ C &= \llbracket \textit{cond} \rrbracket \\ A &= \mathcal{A}(\textit{modname}) \\ X_S &= \textit{addprefix}(\textit{instname}, X_A^E) \\ S &= (D \setminus C) \cup \textit{embedset}(X, X_S, D, S_A^E) \\ \Delta &= \{(C, \textit{instname}, \textit{modname})\} \end{aligned}$$

$\llbracket \textit{impt}_1; \dots; \textit{impt}_n \rrbracket_{S\Delta} XDA = (S_1 \cap \dots \cap S_n, \Delta_1 \cup \dots \cup \Delta_n)$

where

$$(S_i, \Delta_i) = \llbracket \textit{impt}_i \rrbracket_{S\Delta} XDA$$

Then, we define the meaning of a module by the following function. It simply looks up the X , D , X^E and D^E components in \mathcal{N} , the Δ component follows from the module imports only, and the S component is taken from the constraints and the imported modules. (Constraints from parent modules are handled at runtime.)

$\llbracket \text{module } \textit{modname}; \textit{expts}; \textit{impts}; \textit{defns}; \textit{ensns} \rrbracket_{S\Delta} \mathcal{A} = (\textit{modname}, A)$

where

$$\begin{aligned} (X, D, X^E, D^E) &= \mathcal{N}(\textit{modname}) \\ S_L &= \llbracket \textit{ensns} \rrbracket XD \\ (S_I, \Delta) &= \llbracket \textit{impts} \rrbracket_{S\Delta} XDA \\ S &= S_L \cap S_I \\ S^E &= \textit{projectset}(X, X^E, S) \\ A &= (X, D, S, \Delta, X^E, D^E, S^E) \end{aligned}$$

$\llbracket \textit{modl}_1; \dots; \textit{modl}_n \rrbracket = \{P_1, \dots, P_n\}$

where

$$P_i = \llbracket \textit{modl}_i \rrbracket_{S\Delta} \mathcal{A}$$

Finally, the meaning of a modular configuration problem posed as a sequence of modules, is given as a function from the module names to 7-tuples containing

information about the variable sequences, domains, solution sets, instantiation conditions, and those parts of them that are relevant for the exported variables. This function is given by:

$$\mathcal{A} = \mu F_A. \llbracket \mathcal{M} \rrbracket_{S\Delta} F_A$$

2.6 Execution

User interaction with the configurator takes place during what we call the execution phase. At this time, the configuration problem has been compiled into the function \mathcal{A} , mapping module names to 7-tuples.

During user interaction, we will represent the configurator state by a 4-tuple (I, A, R, T) , in which T is itself a set of such 4-tuples. This is the way in which we represent the tree of module instances. This tree is always of finite size (the proof is beyond the scope of this paper). The other elements of the tuple are: I , which is the name of the instance; A , which is the 7-tuple that corresponds with the module the instance was created from; and R , which is the set of remaining valid solutions for this instance.

First, we will look at two auxiliary functions which help with instantiation and synchronization, and after that, we will look at the initial state, choices, and update functions that we need to operate on the configuration states.

So we begin with the first auxiliary function, which takes care of instantiation of subinstances. It takes the definition of a module (A), the remaining solution set (R) and the current set of subtrees (T), and returns the new set of subtrees, keeping all current subtrees, and adding all subinstances which are in the set Δ_A , for which the instantiation condition is satisfied, but which do not yet exist in the current set of subinstances. Note that the new instances are created with a full set of solutions, and without any subinstances of their own. This task is left to all functions that use this function. Also note that it does not take a full (I, A, R, T) tuple as an argument - this is because the semantics turn out to be slightly less tedious this way.

$$subtrees(A, R, T) = T \cup \{(I, \mathcal{A}(M), S_{\mathcal{A}(M)}, \emptyset) \mid (C, I, M) \in \Delta_A, R \subseteq C, \forall N \in T. I_N \neq I\}$$

Then we continue with the second auxiliary function, which takes care of the downward synchronization of instances in the instance tree, and of the resulting instantiations by using *subtrees*. That is to say, when the solution set of an instance is made smaller, this function can be used to ensure the consistency of that instance with all its subinstances, and their subinstances, and so on.

$$\begin{aligned}
\text{propagate}(X_P, R_P, (I, A, R, T)) &= (I, A, R', T') \\
\text{where} \\
X_S &= \text{addprefix}(I, X_A^E) \\
R_S &= \text{projectset}(X_P, X_S, R_P) \\
R' &= R \cap \text{embedset}(X_A, X_A^E, D_A, R_S) \\
T' &= \{\text{propagate}(X_A, R', N) \mid N \in \text{subtrees}(A, R', T)\}
\end{aligned}$$

We now turn to the main thrust of the semantics of execution of a modular configuration problem.

The first function we turn our attention to is the *initialstate* function. The *initialstate* function takes a module name, and returns a tuple representing a configuration state. This tuple is a 4-tuple representing the root instance.

$$\begin{aligned}
\text{initialstate}(\text{modname}) &= (I, A, R, T) \\
\text{where} \\
I &= \epsilon \\
A &= \mathcal{A}(\text{modname}) \\
R &= S_A \\
T &= \{\text{propagate}(X_A, R, N) \mid N \in \text{subtrees}(A, R, \emptyset)\}
\end{aligned}$$

Then, we turn to the operation of finding out what the choices are given a particular configuration state. We have two functions for this purpose: *localchoices* and *globalchoices*. One could consider *localchoices* to be an auxiliary function, as it only generates the choices for the root-instance of the tree. It is used by *globalchoices*, which recursively builds up the set of possible choices for the entire instance tree.

$$\begin{aligned}
\text{localchoices}(I, A, R, T) &= \{(x_1, D_1), \dots, (x_n, D_n)\} \\
\text{where} \\
\langle x_1, \dots, x_n \rangle &= X_A \\
D_i &= \{v_i \mid (v_1, \dots, v_n) \in R\}
\end{aligned}$$

$$\begin{aligned}
\text{globalchoices}(I, A, R, T) &= \text{localchoices}(I, A, R, T) \cup \text{subtreechoices} \\
\text{where} \\
\text{subtreechoices} &= \{(I_N \cdot \langle \cdot \rangle \cdot x, D) \mid N \in T, (x, D) \in \text{globalchoices}(N)\}
\end{aligned}$$

And to finish, we consider the *update* functions. There are two versions. The first deals with the case where a simple variable name is given, in which case the current instance needs to be updated, and the change needs to be propagated to the subinstances. The second deals with the case where the variable is a qualified variable. In this case the subinstance it belongs to needs to be updated first, then the current instance needs to resynchronize with that subinstance, and finally all other subinstances need to resynchronize with the current instance.

The idea is that the correct version of the function to be used is to be determined by pattern matching. If both versions match, which happens with shared variables, either can be used, since shared variables are always kept synchronized.

The first version of the *update* function is for the case where the variable name x is a simple name. In this case, it is the current instance that needs to be modified, after which its subtrees are to be synchronized again. The former is done by cutting down R , and then the latter is done by using a combination of the *subtrees* and *propagate* functions.

$$\text{update}(\langle x \rangle, v, (I, A, R, T)) = (I, A, R', T')$$

where

$$\begin{aligned} \langle x_1, \dots, x_n \rangle &= X_A \\ R' &= R \cap \{(v_1, \dots, v_n) \in D_A \mid x_i = v_i\} \\ T' &= \{\text{propagate}(X_A, R', N) \mid N \in \text{subtrees}(A, R', T)\} \end{aligned}$$

The second version of the *update* function is for the case where the variable name x is prefixed with a path. In this case, we take off the front of the name, which is the instance name of the submodule to which this selection is to be routed. We find the subtree N which has this instance name, and update it recursively. When this is done, we need to synchronize the current module with the new subtree N' , and after that, we need to synchronize the other subtrees with the state of the current module.

$$\text{update}(I \cdot \langle \cdot \rangle \cdot x, v, (I, A, R, T)) = (I, A, R', T')$$

where

$$\begin{aligned} \{N\} &= \{N \in T \mid N_I = I\} \\ N' &= \text{update}(x, v, N) \\ X_S &= \text{addprefix}(I_N, X_{A_N}^E) \\ R_S &= \text{projectset}(X_{A_N}, X_{A_N}^E, R_{N'}) \\ R' &= R \cap \text{embedset}(X_A, X_S, D_A, R_S) \\ T' &= N' \cup \{\text{propagate}(X_A, R', N'') \mid N'' \in \text{subtrees}(A, R', T) \setminus \{N\}\} \end{aligned}$$

3 Conclusion

In this paper, we have extended the two-stage approach compiling into BDDs with the notion of modules. We allow for the combining of such modules by means of shared variables, for conditional instantiation of such modules, and for recursion between such modules. For the resulting configuration system, we guarantee the user that she can select any configuration that is valid, none that are not valid, and that she can always find a finite configuration. In the near future, we will complete the modeling of a real-world modular configuration problem. The results of that work will guide us in our further research.

References

1. Sabin, D., Weigel, R.: Product configuration frameworks - a survey. *Intelligent Systems, IEEE* **13** (1998) 42–49
2. Tiihonen, J., Soinen, T., Niemelä, I., Sulonen, R.: Empirical testing of a weight constraint rule based configurator. In: *ECAI 2002 Configuration Workshop*. (2002) 17–22 <http://www.soberit.hut.fi/pdmg/Empirical/index.html>.
3. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. *Artificial Intelligence* **1-2** (2002) 199–234 <ftp://fpt.irit.fr/pub/IRIT/RPDMP/Configuration/>.
4. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: *Proceedings of the International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems*, DTU-tryk (2004) 131–138