Enfragmo: A System for Modelling and Solving Search Problems with Logic

Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, David Mitchell

Simon Fraser University
{aaa78,xwa33,sta44,ter,mitchell}@cs.sfu.ca

Abstract. In this paper, we present the Enfragmo system for specifying and solving combinatorial search problems. It supports natural specification of problems by providing users with a rich language, based on an extension of first order logic. Enfragmo takes as input a problem specification and a problem instance and produces a propositional CNF formula representing solutions to the instance, which is sent to a SAT solver. Because the specification language is high level, Enfragmo provides combinatorial problem solving capability to users without expertise in use of SAT solvers or algorithms for solving combinatorial problems. Here, we describe the specification language and implementation of Enfragmo, and give experimental evidence that its performance is comparable to that of related systems.

1 Introduction

Computationally hard search and optimization problems are ubiquitous in science, engineering and business. Examples include drug design, protein folding, phylogeny reconstruction, hardware and software design, test generation and verification, planning, timetabling, scheduling and so on. In rare cases, practical application-specific software exists, but most often development of successful solution methods requires specialists to apply technology such as mathematical programming of constraint logic programming systems, or develop custom-refined implementations of general algorithms, such as branch and bound, simulated annealing, or reduction to SAT.

One goal of development of the Enfragmo system [1] is to provide another practical technology for solving combinatorial search problems, but one which would require considerably less specialized expertise on the part of the user, thus making technology for solving these problems accessible to a wider variety of users. In this approach, the user gives a precise specification of their search (or optimization) problem in a high-level declarative modelling language. A solver then takes this specification, together with an instance of the problem, and produces a solution to the problem (if there is one).

A natural formalization of search problems in general is as model expansion (MX) [2], which is the logical task of expanding a given structure by new relations. Formally, users axiomatize their problems, formalized as model expansion, in some extension of classical logic. A problem instance, in this formalization, is a finite structure, and solutions to the instance are expansions of this structure that satisfy the specification formula. At present, our focus is on problems in the complexity class NP. For this case, the specification language is based on classical first-order logic (FO). Fagin's theorem [3] states that the problems which can be axiomatized in the existential fragment of second order logic ($\exists$SO) are exactly those in NP, and thus the problems which can be axiomatized as FO MX are exactly the NP search problems.

Enframgo's operation is based on grounding, which is the task of producing a variable-free first-order formula representing the expansions of the instance structure which satisfy the specification formula – in other words, the solutions for the instance. The ground formula is mapped to a propositional CNF formula, which is sent to a SAT solver. For any fixed FO formula, grounding can be carried out in polynomial time, so grounding provides a universal polytime reduction to SAT for problems in NP. (Developing grounding to other languages, for example to use SMT solvers, is a promising direction which we have begun exploring.) An important advantage in solving through grounding and transformation to SAT, or other standard ground language, is that the performance of ground solvers is constantly being improved, and we can always select from the best solvers available.

Many interesting real-world problems cannot be conveniently expressed in pure FO MX, in particular if their natural descriptions involve arithmetic or inductive properties. Examples of the former include Knapsack and other problems involving weights or costs, while examples of the latter include the Traveling Salesman problem and other problems involving reachability. To address these issues, Enfragmo's specification language includes a limited use of inductive definitions, and extends classical first order logic with arithmetic and aggregate operators. The authors of [2] emphasized the importance of having inductive definitions in a specification language. In the case of arithmetic, care must be exercised to avoid increasing the expressive power of the language beyond NP, which would prevent polytime grounding to SAT. A theoretical investigation of the issues involved appears in [4–6]. Extension of the basic grounding algorithm of Enfragmo to arithmetic terms, including aggregate operators, is described in [7].

2 Specification Language

The specification language of the Enfragmo system is based on multi-sorted classical first-order logic extended with inductive definitions, arithmetic functions, and aggregate operators. We will illustrate the language with two examples, and give brief discussions of some major features. The examples use the actual ASCII input representation. The mapping from logical symbols to ASCII symbols used is given in Table 1. For the full description of the input language, please refer to the Enfragmo manual, which is available from [1].

An Enfragmo specification consists of four main sections, delineated by keywords. The `GIVEN:` section defines the types and vocabulary used in the specification. The `FIND:` section identifies the vocabulary symbols for which the solver must find interpretations, that is, the functions and relations which will constitute a solution. Interpretations of the remaining vocabulary symbols are given by the problem instance. The third part consists of one or more `PHASE:` sections, each of which contains an optional

Table 1. ASCII Equivalents for Logical Symbols

Logical Symbol	$\forall$	$\exists$	$\wedge$	$\vee$	$\neg$	$\rightarrow$	$\leftrightarrow$
ASCII Representation	!	?	&	|	~	=>	<=>

```
GIVEN:
   TYPES: Vtx Clr;
   PREDICATES: Edge(Vtx,Vtx), Colour(Vtx,Clr);
FIND: Colour;
   // expansion predicate(s) are listed under FIND
   //(instance predicates are those that are not expansion)
PHASE:
 SATISFYING:
   // every vertex has at least one colour
   !v:Vtx : ?c:Clr : Colour(v,c);
   // no vertex has more than one colour
   !v:Vtx c:Clr : Colour(v,c) => ~?c2:Clr < c : Colour(v,c2);
   // no two vertices of the same colour are adjacent
   !u:Vtx v:Vtx c:Clr : Colour(u,c) & Colour(v,c)=> ~Edge(u,v);
PRINT: Colour; // solution can be printed
```

Fig. 1. Enfragmo specification of K-colouring.

`FIXPOINT:` part, which provided an inductive definition, followed by a `SATISFYING:` part, which consists of a set of sentences in the extended first order logic. If there are multiple `PHASE:` sections, the define a sequence of expansions. One way such a sequence can be used is to carry out a kind of pre-processing or post-processing, which may support more convenient axiomatizations or more efficient solving. An example is provided in the section on inductive definitions below. Finally, the `PRINT:` section identifies relations that are to be displayed, if a solution is found.

Example 1 (Graph K-Colouring). Graph colouring is a classic and well-studied NP-hard search problem. The task is to colour vertices of a given graph using colours from a given set of K colours, so that no two adjacent vertices have the same colour. To axiomatize this problem, we introduce two sorts, vertices and colours. The axiomatization says that there is a binary relation Colour which must be a proper colouring of the vertices. The corresponding Enfragmo specification is given in Figure 1.

Arithmetic and Aggregates Enfragmo specifications have two kinds of types: integer types and enumerated types. Terms of integer types may use the arithmetic functions $+$, $-$, $*$, and $ABS(\cdot)$, which have their standard meaning for the integers. Arithmetic terms also include the aggregate operators maximum, minimum, sum, and cardinality. In the following, if $\phi(\bar{x})$ is a formula with free variables $\bar{x}$, then $\phi^{\mathcal{B}}[\bar{a}]$ denotes the truth value of ϕ in structure $\mathcal{B}$ when the variables $\bar{x}$ denote the domain elements $\bar{z}$, and similarly for terms $t(\bar{x})$. The aggregate terms are defined, as follows, with respect to a structure $\mathcal{B}$ in which the formula containing the term is true.

$Max_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_M\}$ denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the maximum value obtained by $t^{\mathcal{B}}[\bar{a},\bar{b}]$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true, or d_M (the default value) if there are none.

$Min_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m\}$ is defined dually to Max.

$Sum_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the sum of all values $t^{\mathcal{B}}[\bar{a},\bar{b}]$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true.

$Count_{\bar{x}}\{\phi(\bar{x},\bar{y})\}$ denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the number of tuples $\bar{a}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true.

Example 2 illustrates use of arithmetic terms, including sum and count aggregates.

Example 2 (A Knapsack Problem Variant). Consider the following variation of the knapsack problem: We are given a set of items (loads) $L = \{l_1, \cdots, l_n\}$, each with an integer weight $W(l_i)$, and m knapsacks $K = \{k_1, \cdots, k_m\}$. The task is to put the n items into the m knapsacks, while satisfying the following constraints. 1) Certain items must go into preassigned knapsacks, as specified by the binary instance predicate P; 2) H of the m knapsacks are high capacity, and can hold items with total weight Cap_H, while the remainder have capacity Cap_L; 3) No knapsack may contain two items with weights that differ by more than D. Each of Cap_H, Cap_L and D is an instance function with arity zero, i.e. a given constant. An Enfragmo specification for this problem is given in Figure 2. Q is the mapping of items to knapsacks that must be constructed.

Inductive Definitions Enfragmo supports a limited use of inductive definitions. In the current implementation, the open predicates [2] in a definition must be instance predicates, or have been constructed explicitly in a previous `Phase:` section. Even this limited form has proved to be very useful in practice. These definitions can be used to efficiently compute useful information, such as a bound or partial solution, which can be used later to help solve a problem more efficiently. For example, Graph Colouring can be solved more efficiently by having inductive definitions in an initial group of phases compute a maximal clique in the graph, and pre-assign distinct colours to the vertices in that clique. (A further improvement might be to construct a maximal clique containing a vertex of maximum degree.) Then a final phase can construct a colouring of the graph restricted by the pre-computed colouring of the large clique. Use of these inductive definitions can also support writing of natural axiomatizations, by allowing the introduction of defined terms without incurring a performance penalty.

3 Implementation

Enfragmo's input is a problem specification, stated in the language described in Section 2, together with a description of an instance. (The syntax for specifying instances is described in the manual. Eventually, instances may be retrieved by queries to a database or other source.) The phases in the specification are solved one-by-one, in the order written. For each phase, any predicates defined by an inductive definition are computed, and then the satisfying phase is solved by grounding. This in turn involves three stages:

```
GIVEN:
   TYPES: Item Knaps;
   INTTYPES: Weight ItemCount;
   PREDICATES: P(Item, Knaps) Q(Item, Knaps);
   FUNCTIONS:
     W(Item): Weight
     Cap_H(): Weight
     Cap_L(): Weight
     D(): Weight
     H(): ItemCount;
FIND: Q;
PHASE:
 SATISFYING:
   // Q is a function mapping items to knapsacks
   !l:Item : ?k:Knaps : Q(l, k);
   !l:Item k1:Knaps k2:Knaps : Q(l, k1) & Q(l, k2) => k1 = k2;
   // Q agrees with the pre-assignment P
   !l:Item k:Knaps : P(l, k) => Q(l, k);
   // The total weight in each knapsack is at most Cap_H
   !k:Knaps: SUM{l:Item; W(l); Q(l, k)} <= Cap_H();
   // At most H knapsacks have total weight greater than Cap_L
   COUNT{k:Knaps; SUM{l:Item; W(l); Q(l, k)} > Cap_L()} <= H();
   // Items in a knapsack differ in weight by at most D
   !k:Knaps l1:Item l2:Item : Q(l1, k) & Q(l2, k)
                                => ABS(W(l1) - W(l2)) <= D();
PRINT: Q;
```

Fig. 2. Enfragmo specification for Knapsack variant.

1) grounding each formula with respect to the instance to produce a ground FO formula representing the solutions; 2) the ground formula is transformed to a propositional CNF formula; 3) a SAT solver is called on the CNF formula; 4) if the SAT solver reports a satisfying assignment, it is mapped back to a description of a solution in the vocabulary of the specification.

Grounding Enfragmo computes a grounding of a formula bottom up, in a process analogous to bottom up evaluation of a database query using the relational algebra. Here, an extension of the relational algebra is used, in which an formula is associated with each tuple. A tuple contains domain elements, and the associated formula is (equivalent to) a ground instance of a sub-formula of the specification formula, with variables instantiated by constants denoting the domain elements in the tuple. An "answer" to a sub-formula of the specification formula is an extended table representing all instantiations of the sub-formula. Similarly, and answer to a terms is a set of triples, each consisting of an instantiation of the arguments, a value the term may denote, and a formula. Details can be found in [7] and [8]. The answer for a sentence consists of an empty tuple associated with a formula that is a grounding of the sentence with respect to the instance.

Efficiency of this grounding method requires using suitable data structures. For efficiency, all formulas in computed answers are represented in a dag which is constructed as the operations of the algebra are applied. Next we briefly describe some aspects of the implementation of tables representing answers.

In the tables representing answers to formulas, it is natural have non-existence of a tuple correspond to associating the formula False with the tuple. However, negating a sparse table with this convention produces a very dense table in which many tuples are associated with the formula True. To help keep tables sparse, we employ two kinds of tables, one in which absence of a tuple corresponds to False, and one where it corresponds to True. Details on design and performance of True/False tables are given in [8]. It is often the case that, in an answer for a sub-formula, the instantiated formulas are independent of the instantiations of some of the free variables. In this case, the table explicitly records only the partial instantiations that are needed. This method is described in [9] and [8] as tables with "hidden variables".

A simple term is a term whose denotation can be computed (with respect to an instance), just using the assignments to its free variables. For example, $W(l)$ in Example 2 is a simple term. The formula for each tuple in an answer to a simple term is either True or False. A term which is not simple is called complex. The Count aggregate used in Example 2 is a complex term, because its value depends on the expansion predicate Q (the solution). To represent the answer to a complex term occurring as an argument to sub-formula ϕ, we have two data-structures: 1) A hash-map which maps each value o which term t may denote to a table which is an answer to the formula $t(\bar{x}) = o$, and 2) A table which can be viewed as being the answer to the formula $\phi(\bar{x}, y) : t(\bar{x}) = y$. Methods for efficiently constructing the answers to complex terms, such as terms containing nested count and sum aggregates, are examined in [7] and [10].

CNF Transformation The set of answers for the sentences of a specification are then transformed to a propositional CNF formula. As usual, this is done using a refinement of Tseitin's polytime transformation to CNF [11]. Refinements include re-writing the formula into negation normal form, so negations occur only on atoms; flattening nested conjunctions and disjunctions; and in some cases merging of identical sub-formulas.

4 Experimental Evaluation

In this section, we compare the performance of Enfragmo to other grounding-based systems. A set of NP-hard of problems were chosen from [12]. We excluded problems for which all instances in the collection are easy. We also excluded problems where the sum aggregate is central, as the current implementation of sum in Enfragmo is preliminary and does not perform well. (We are currently studying better methods for sum. For example, see [13]). The other solvers are Clingo (v 3.0.3) [14], DLV (v 2010-10-14) [15], and IDP (v 2.20) [16]. For each system, we used specifications provided by the system authors, obtained from [12]. The experiments were run on an Intel Xeon L5420 quad-core 2.5 GHz processor, with a timeout of 600 seconds. All specifications, instances, and scripts used for the experiments can be downloaded from [1]. The results are given in Table 2. The entry n/t indicates that n instances were solved, each within the 600 second timeout, in a total time of t seconds. The time t includes the time for all the runs that timed out.

Table 2. Performance comparison of Enfragmo and other systems. The entry n/t means that n instances were solved in total time of t seconds. The 600-second timeouts are included in the times. The best result for each problem is in bold.

Problem	# of Instances	Clingo	DLV	IDP	Enfragmo
GraphColoring	29	9/12400	8/13398	9/12199	**27/6965**
HamiltonianPath	29	**29/1.6**	20/6856	29/2.1	29/308
SchurNumbers	29	29/889	18/8273	28/1452	**29/643**
BlockedNQueens	29	**29/165**	28/9870	29/896	29/1278
ConnectedDominatingSet	20	**20/969**	13/6190	17/3258	19/2038
DisjunctiveScheduling	10	10/1174	5/3581	10/1008	**10/421**
Total	146	126/15601	92/48170	122/18818	**143/11656**

Table 2 shows that Enfragmo was able to solve almost all the instances in the collection, and performed the best on three of the six problems. Enfragmo also performed the best by the aggregate measures of total number of instances solved and total time spent.

5 Conclusion

We presented the Enfragmo system for modelling and solving combinatorial search problems. It provides users with a convenient way to specify and solve computationally hard problems, in particular search problems whose decision versions are in the complexity class NP. The performance of the Enfragmo system is comparable to that of related systems.

References

1. `http://www.cs.sfu.ca/research/groups/mxp/`.
2. Mitchell, D., Ternovska, E.: A framework for representing and solving NP search problems. In: Proc. AAAI. (2005) 430–435
3. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. Complexity of Computation (1974) 43–74
4. Ternovska, E., Mitchell, D.: Declarative programming of search problems with built-in arithmetic. In: Proc. of IJCAI. (2009) 942–947

5. Tasharrofi, S., Ternovska, E.: Built-in arithmetic in knowledge representation languages. In: NonMon at 30 (Thirty Years of Nonmonotonic Reasoning). (October 2010)
6. Tasharrofi, S., Ternovska, E.: PBINT, a logic for modelling search problems involving arithmetic. In: Proc. LPAR-17, Springer (October 2010) LNCS 6397.
7. Aavani, A., Wu, X.N., Ternovska, E., Mitchell, D.G.: Grounding formulas with complex terms. In: Canadian AI, the 24th Canadian Conference on Artificial Intelligence. (2011) 13–25
8. Aavani, A., Tasharrofi, S., Ünel, G., Ternovska, E., Mitchell, D.G.: Speed-up techniques for negation in grounding. In: LPAR-16. (2010) 13–26
9. Mohebali, R.: A method for solving NP search problems based on model expansion and grounding. Master's thesis, Simon Fraser University (2006)
10. Aavani, A., Wu, X., Mitchell, D., Ternovska, E.: Grounding Cardinality Constraints. LPAR-16 short paper (2010)
11. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic (1968) 115–125
12. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. LPNMR (2009) 637–654
13. Aavani, A.: Translating Pseudo-Boolean constraints into CNF. Theory and Applications of Satisfiability Testing (SAT) (2011) 357–359
14. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Commun. 24(2) (2011) 105–124
15. Dell'Armi, T., Faber, W., Ielpa, G., Koch, C., Leone, N., Perri, S., Pfeifer, G.: System description: DLV. In: LPNMR. (2001) 424–428
16. Wittocx, J., Mariën, M., Denecker, M.: The IDP system: A model expansion system for an extension of classical logic. In: Proceedings of the 2nd Workshop on Logic and Search. (2008) 153–165