Implementation of Propagation-Based Constraint Solver in IMS

Igor Ol. Blinov

Kherson State University, 27, 40 rokiv Zhovtnya St., Kherson, Ukraine 73000

anubis.igor@gmail.com

Abstract. Article compiling the main ideas of creating propagation-based constraint solver, theoretical basis of constraint programming its implementation in IMS (Insertion Modeling System) and creating prototype of IMS constraint solver.

Keywords. IMS, constraint programming, solvers

Key terms. Mathematical model, process, research

1 Introduction

In today's world, computing problems are becoming more and more applicable in real life. Some problems arise from the real problem of production requirements of science, etc.

The challenge of creating a real-life computational equivalent of the human mind requires that we had better understand at a computational level how natural intelligent systems develop their cognitive and learning functions. The narrow focus of science on this challenge brings together four schools of thought:

1. Computational neuroscience, that tries to understand how the brain works in terms of connectionist models;
2. Cognitive modeling, pursuing higher-level computational description of human cognition;
3. Human-level artificial intelligence, aiming at generally intelligent artifacts that can replace humans at work;
4. Human-like learners: artificial minds that can be understood by humans intuitively, that can learn like humans, from humans and for human needs.

A solution to the problems of this type consists of several parts. The main parts are the tools and methodology. Tools provided at this time to address them varied. We consider the approach of cognitive modeling and cognitive architectures.

Important part of cognitive architecture – solvers. For creating solvers, we use the constraint programming method.

Let’s consider this method with its application area and specification.

1.1 Basic Concepts

Constraint programming is a powerful method for solving combinatorial (optimization) problems, which has proven effective and efficient in a wide range of application areas.

Constraint programming is an embedding of constraints in a host language. The first host languages used were logic programming languages, so the field was initially called constraint logic programming. The two paradigms share many important features, like logical variables and backtracking. Today most Prolog (for example) implementations include one or more libraries for constraint logic programming.

The difference between the two is largely in their styles and approaches to modeling the world. Some problems are more natural (and thus, simpler) to write as logic programs, while some are more natural to write as constraint programs.

The constraint programming approach is to search for a state of the world in which a large number of constraints are satisfied at the same time. A problem is typically stated as a state of the world containing a number of unknown variables. The constraint program searches for values for all the variables.

Application areas. Many hard, real-world combinatorial problems lend themselves to modeling as constraint satisfaction or optimization problems. The Handbook of Constraint Programming (Rossi et al., 2006) lists example applications in the areas of scheduling and planning, vehicle routing, configuration, networks (such as power or pipeline networks), and bioinformatics. Further application areas include computational linguistics (for example Duchier, 1999), as well as verification (Yuan et al., 2006) and optimization (van Beek and Wilken, 2001) of computer programs.

Constraint Satisfaction Problems. A combinatorial problem is modeled as a set of variables, representing the objects the problem deals with, and a set of constraints, representing the relationships among the objects. Such a combinatorial problem is called a *Constraint Satisfaction Problem (CSP)*. The common case where the variables can only take values from a finite universe is called a finite domain constraint satisfaction problem. A constraint programming system implements variables and constraints and provides a solution procedure for CSPs, which tries to find an assignment to the variables that satisfies all of the constraints. Clearly, solving CSPs is NP-hard in general, as the satisfiability of Boolean formulas (SAT) is one instance.

As we use the constraint programming approach, solvers are called *constraint solvers*.

Constraint solvers. The success of constraint programming as a field is due to the availability of effective and efficient solution procedures that can solve these practical problems. This paper concentrates on finite-domain constraint programming, implemented in a propagation-based constraint solver, based on exhaustive search. This class of solvers has been successful because of its best-of-several worlds approach. They combine classic AI search methods with advanced implementation techniques from the Programming Languages community and efficient algorithms from Operations Research. Furthermore, the Constraint Programming community has identified global constraints as an important tool to make the structure of constraint problems explicit and achieve strong propagation. Dedicated propagation algorithms for many different global constraints are available.

We consider the variety of solvers, which are called propagation-based constraint solvers.

Propagation-based constraint solving. At the heart of a propagation-based constraint solver, propagators realize the constraints of a CSP by pruning the variable domains. A propagator removes values from variable domains that cannot be part of any solution of its constraint. Propagators for particular constraints are usually implemented as specialized algorithms. The constraint solver computes a fixed point of all propagators, maximizing the amount of inference they can contribute. It then splits the problem and solves the resulting smaller problems recursively.

This process of inference is called **constraint propagation.** As the main inference method in constraint programming systems, constraint propagation infers that certain values cannot be part of certain variable domains any more because they violate some constraint. The entities that perform constraint propagation are called propagators.

Constraint satisfaction problems are modeled with respect to a finite set of variables X and a finite set of values V. We typically write variables as $x,y,z \in X$, and refer to values as v, w V.

For article example CSP, we choose the *problem of scheduler generation.* The atom of schedule system is one record of type {teacher, group, subject, room}. Let's take this simple set for example. We will describe each part of article using this example, specifying and describing a detail.

1.2 Assignments and Constraints

Constraint satisfaction problem solution should provide a unique correspondence between the values and the variables. A constraint restricts which assignments of values to variables are allowed. Next definition captures assignments and constraints.

Definition 1 An assignment *a* is a function mapping variables to values. The set of all assignments is Asn := X → V. A constraint *c* is a set of assignments, $c \in$ Con := P (Asn) = P (X → V) (we write P (S) for the power set of S). It corresponds to a relation over the variables in X. Any assignment a Є c is a solution of c.

In basic works of Guido Tack [1], researchers base constraints on full assignments, defined for all variables in X. However, for typical constraints, only a subset vars(c) of the variables is significant; the constraint is the full relation for all $x \in \text{vars}(c)$. More formally, a constraint c is the full relation for a variable x if and only:

$$\forall v \in V, \forall a \in c : a[v/x] \in c\,, \qquad (1)$$

where a[v/x] is the assignment a′ where a′(x) = v and a′(y) = a(y) for all variables y ≠ x.

Consequently, the significant variables of **c** can be defined as

$$\text{vars}(c) := \{x \in X \mid \exists v \in V; \exists a \in c : a[v/x] \notin c\} \qquad (2)$$

Constraints are either written as sets of assignments, or just stated as mathematical expressions with the usual meaning. We use the notation [[·]] when we want to stress that we mean the constraint; for example, we write [[x < y]] to denote the constraint {a ∈ Asn a(x) < a(y)} .

Using IMS we will define our constraints as a part of function that preceded the insertion function, as we see later. It should done its work before insertion will, or return the forbiddance for insertion. Alternatively, like a part of insertion function, with one more conditional expression.

1.3 Domains and Constraint Satisfaction Problems. Propagators

Constraints represent one half of the Constraint Satisfaction Problem solutions. The other part is the initial set of values that each variable can take. For example in a Golf Club schedule, each variable must take a value from the set of golf-players. In our example, variables will take the value from the set of triples <Teacher, Group, Subject>. A mapping from variables to sets of possible values is a domain. Some popular domains for constraint programming are:

- Boolean domains, where only true/false constraints apply (SAT problem)
- Integer domains, rational domains
- Linear domains, where only linear functions are described and analyzed (although approaches to non-linear problems do exist)
- Finite domains, where constraints are defined over finite sets
- Mixed domains, involving two or more of the above

Finite domains are one of the most successful domains of constraint programming. In some areas (like operation research), constraint programming is often identified with constraint programming over finite domains.

Definition 2 A domain **d** is a function mapping variables to sets of values, such that d(x) V. The set of all domains is Dom: = X → P (V). The set of values in d for a particular variable x, d(x), is called the variable domain of x. A domain d represents a set of assignments, a constraint, defined as

$$con(d) := \{a \in Asn \mid \forall x \in X : a(x) \in d(x)\} \qquad (3)$$

Said that an assignment $a \in con(d)$ is licensed by d.

In our example, we can implement two types of domain realization. Each domain can be realized as a state of an agent, and be (or not) omitted by propagator during insertion, or other way – store all sets of domain in environment' state.

Definition 3 A constraint satisfaction problem (CSP) is a pair <d, C> of a domain d and a set of constraints C. The constraints C are interpreted as a conjunction of all c Є C and are thus equivalent to the constraint $\{a \in Asn \mid \forall c \in C : a \in c\}$. The solutions of a CSP <d, C> are the assignments licensed by **d** that satisfy all constraints in C, defined as

$$sol(< d,C >) := \{a \in con(d) \mid \forall c \in C : a \in c\} \qquad)$$

2 Propagators

The basis of a propagation-based constraint solver is a search procedure, which systematically enumerates the assignments licensed by the domain d of a CSP <d, C>.

For each assignment, the solver uses a decision procedure for each constraint to determine whether the assignment is a solution of the CSP. Enumerating all assignments would be infeasible in practice, so in addition to the *decision procedure*, the solver employs a *pruning procedure* for each constraint, which may rule out assignments that are not solutions of the constraint.

Two problems, decision, and pruning procedure for constraints implemented by propagators. Each propagator induces particular constraint. Propagator decides for a given assignment, whether it satisfies the induced constraint, and it can cut off (prune) these tasks from the domain that do not satisfy the constraint. Interleaving propagation and search yield sound and complete procedure for solving CSP. It is complete, because only the assignments that are not solutions are pruned by propagators, and all other assignments are in enum. This is sound, because for each of these tasks, the propagators to decide whether it's the definition of solution. The formal definition of propagators author (see [1]), reflects the minimum properties that are needed in order to get a sound and complete solver. Thus, this model differs from the one commonly found in the literature. Furthermore, knowledge of the unique characteristics of the propagators induced constraints is new. The authors define the propagators in terms domains.

A propagator is a function **p** that takes a domain as its argument and returns a stronger domain, it may only prune assignments. If the original domain was an assigned domain {a}, the propagator either accepts it (p({a}) = {a}) or rejects it (p({a}) = 0), realizing the decision procedure for its constraint. In fact, each propagator induces a unique constraint, the set of assignments that it accepts. To make this setup work, we need one additional restriction. The decision procedure and the pruning procedure must be consistent: if the decision procedure accepts an assignment, the pruning procedure must never remove this assignment from any domain—this property is called soundness.

Definition 4 A propagator is a function $p \in Dom \to Dom$ that is:

- Contracting: $p(d) \subseteq d$ for any domain d
- Sound: for any domain $d \in Dom$ and any assignment $a \in Asn$, if $\{a\} \subseteq d$, then $p(\{a\}) \subseteq p(d)$

The set of all propagators is Prop. If a propagator p returns a strictly stronger domain $(p(d) \subset d)$, we say that p prunes the domain d. The propagator p induces the constraint c_p defined by the set of assignments accepted by p:

$$c_p := \{a \in Asn \mid p(\{a\}) = \{a\}\} \tag{5}$$

Soundness expresses exactly that the decision and the pruning procedure realized by a propagator are consistent. A direct consequence is that a propagator never removes assignments that satisfy its induced constraint.

Focusing on our problem, we implement the idea of propagators in additional functions that will proceed the domains (as agent state or environment state) before insertion. Then after the insertion call other propagators to prune from their induced domain unnecessary values to decreasing with each step the search field.

For abstracting the solution of the problem we should give the definition and describing of *propagation problem,* as a higher model of solution of problems of given type.

2.1 Propagation Problem

Propagators were defined as a refinement of constraints – each propagator induces one particular constraint, but in addition has an operational meaning, its pruning procedure. It is possible to define the operational equivalent of a CSP, a propagation problem. Propagation problems realize all constraints of a CSP using propagators.

Definition 5 A propagation problem (PP) is a pair <d, P> of a domain *d* and a set of propagators P. The induced constraint satisfaction problem of a propagation problem <d, P> is the CSP $< d,\{c_p \mid p \in P\} >$. The solutions of a PP <d, P> are the solutions of the induced CSP, $sol(< d,P >) := sol(< d,\{c_p \mid p \in P\} >)$.

The set of solutions of a PP d, P can be defined equivalently as $sol(< d,P >) := \{a \in Asn \mid \forall p \in P : p(\{a\}) = \{a\}\}$, just applying the definitions of induced constraints and solutions of CSPs.

Solution of propagation problem make by using propagators, at each step of inserting an agent into environment. For this, as we mentioned earlier, we will inspect by propagator each domain that is stored in the attributes of the agent. Before each insertion, a domain stored in the attributes of the agent will checked by parameters gained while working. Let’s we look at insertion machine architecture

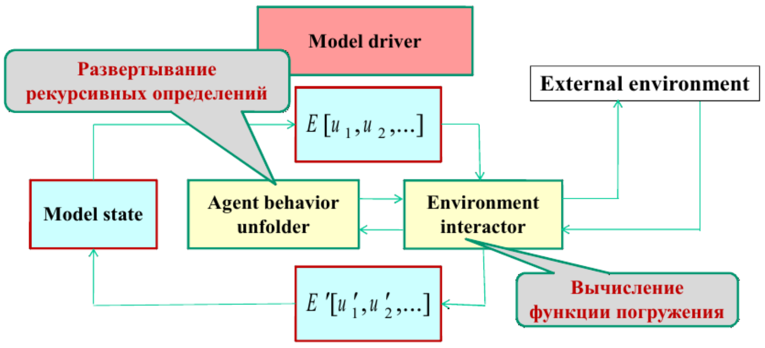

Fig. 1. Insertion machine architecture

The calculation of a insertion function can be tested for the ability to insert the propagator of the agent, which is inserted. If propagator exhausted domain that store in the agent, inserting step will rejected.

Existence of strongest and weakest propagators. Propagators combine a decision procedure with a pruning procedure. While the decision procedure determines the constraint a propagator induces, there is some liberty in the definition of the pruning, as long as it is sound. Thus, there are different propagators for the same constraint, and they can be arranged in a partial order according to their strength:

Definition 6 Let p1 and p2 be two propagators that induce the same constraint. Then p1 is stronger than p2 (written $p_1 \subseteq p_2$) if and only if for all domains d, $p_1(d) \subseteq p_2(d)$.

2.2 Propagation as a Transition System

Propagation as a Transition System. A propagation-based solver interleaves constraint propagation and search, where constraint propagation means to prune the domain as much as possible using propagators, before search resorts to enumerating the assignments in the domain. Propagating as much as possible means, in the context of propagation problems, to compute a mutual fixed point of all propagators.

Transitions. Let <d, P> be a propagation problem. If there is a propagator $p \in P$ that can prune the domain d, that is, if $p(d) \subset d$, then applying p yields a new, simpler propagation problem, <p(d), P> . Soundness of p makes sure that the new problem has the same set of solutions as the original problem, $sol(< d,P >) = sol(< p(d),P >)$.

A propagation problem thus induces a transition system, where a transition is possible from a domain d to a domain $d' \subset d$ if there is a propagator $p \in P$ such that p(d) = d′ . Written such a transition

$$d \mid -p \rightarrow d' \tag{6}$$

Definition 7 Let d be a domain. A transition $d \mid -p \rightarrow d'$ with a propagator p to a domain d′ is possible if and only if d′ = p(d) and $d' \subset d$. The transition system of a propagation problem <d, P> consists of all the transitions that are possible with propagators $p \in P$, starting from d. A terminal domain, that is, a domain d such that there is no transition $d \mid -p \rightarrow p(d)$ for any propagator $p \in P$, is called stable.

Written $d \Rightarrow d'$ if there is a sequence of transitions that transforms d into a stable domain d′. This sequence is empty, $d \Rightarrow d$, if d is stable.

The transition system of a propagation problem is non-deterministic, as there are many possible chains of propagation that result in a stable domain.

Fixed points. The important theorem that ensures that constraint propagation is useful in practice is that, given a propagation problem <d,P>, its transition system is finite and terminating. No matter in what order the propagators are applied, we reach a stable propagation problem after a finite number of steps.

The naive approach to solving a propagation problem <d,P> is to generate all assignments $a \in d$, and then use the propagators $p \in P$ to check whether a satisfies all constraints. This approach makes use of the fact that propagators realize decision procedures for their induced constraints, but does not use their pruning capabilities. A solver that proceeds naively in this fashion is said to follow the *generate-and-test approach.*

3 Conclusion

In conclusion, it can be said that the search for solutions by the *generate-and-test approach* is inefficient, so we will consider other options. Nevertheless, this option works well for prototyping, because of ease of implementation. In the future, we plan to create a working prototype of the university schedule, which plans to make a universal, independent of the input parameters, the types of activities and a list of lessons. The most effective solution to this problem now is supposed to use multi-layer environments, for pruning each of input domains by few environments and few propagators.

References

1. Tack, G.: Constraint Propagation. Models, Techniques, Implementation. Saarbrucken (2009)
2. Letichevsky, A., Letichevskyi, O., Peschanenko, V., Blinov, I., Klionov, D.: Insertion Modeling System and Constraint Programming. In: Ermolayev, V. et al. (eds.) Proc. 7th Int. Conf. ICTERI 2011, Kherson, Ukraine, CEUR-WS vol. 716 (2011)
3. Gilbert, D.R., Letichevsky, A.A.: A Universal Interpreter for Nondeterministic Concurrent Programming Languages. Fifth Compulog Network Area Meeting on Language Design and Semantic Analysis Methods (1996)
4. Letichevsky, A., Gilbert, D.: A General Theory of Action languages. Cybernetics and System Analyses, l(1), 16–36 (1998)
5. Letichevsky, A., Gilbert, D.: A Model for Interaction of Agents and Environments. In: D. Bert, C. Choppy, P. Moses, (eds.): Recent Trends in Algebraic Development Techniques. LNCS 1827, pp. 311–328, Springer Verlag, Berlin Heidelberg (1999)
6. Letichevsky, A.: Algebra of Behavior Transformations and its Applications. In: Kudryavtsev, V.B., Rosenberg, I.G. (eds) Structural theory of Automata, Semigroups, and Universal Algebra, NATO Science Series II. Mathematics, Physics and Chemistry, Springer , vol. 207, pp. 241–272 (2005)
7. Martin, G., Selic, B. (eds.): UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, Amsterdam (2003)
8. Letichevsky, A., Kapitonova, J., Letichevsky, A. Jr., Volkov, V., Baranov, S., Kotlyarov, V., Weigert, T.: Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 662–675 (2005)
9. Kapitonova, J., Letichevsky, A., Volkov, V., Weigert, T.: Validation of Embedded Systems. In: Zurawski, R. (ed.) The Embedded Systems Handbook, CRC Press, Miami (2005)
10. Letichevsky, A., Kapitonova, J., Volkov, V., Letichevsky, A. Jr., Baranov, S., Kotlyarov, V., Weigert, T.: System Specification with Basic Protocols. Cybernetics and System Analyses, 4, 479–493 (2005)