Artificial Intelligence 211 (2014) 1–33

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Generating custom propagators for arbitrary constraints

Ian P. Gent, Christopher Jefferson *, Steve Linton, Ian Miguel, Peter Nightingale *

School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK

ARTICLE INFO

Article history:
Received 29 November 2012
Received in revised form 27 February 2014
Accepted 2 March 2014
Available online 12 March 2014

Keywords:
Constraint programming
Constraint satisfaction problem
Propagation algorithms
Combinatorial search

ABSTRACT

Constraint Programming (CP) is a proven set of techniques for solving complex combinatorial problems from a range of disciplines. The problem is specified as a set of decision variables (with finite domains) and constraints linking the variables. Local reasoning (*propagation*) on the constraints is central to CP. Many constraints have efficient constraint-specific propagation algorithms. In this work, we generate custom propagators for constraints. These custom propagators can be very efficient, even approaching (and in some cases exceeding) the efficiency of hand-optimised propagators.

Given an arbitrary constraint, we show how to generate a custom propagator that establishes GAC in small polynomial time. This is done by precomputing the propagation that would be performed on every relevant subdomain. The number of relevant subdomains, and therefore the size of the generated propagator, is potentially exponential in the number and domain size of the constrained variables.

The limiting factor of our approach is the size of the generated propagators. We investigate symmetry as a means of reducing that size. We exploit the symmetries of the constraint to merge symmetric parts of the generated propagator. This extends the reach of our approach to somewhat larger constraints, with a small run-time penalty.

Our experimental results show that, compared with optimised implementations of the table constraint, our techniques can lead to an order of magnitude speedup. Propagation is so fast that the generated propagators compare well with hand-written carefully optimised propagators for the same constraints, and the time taken to generate a propagator is more than repaid.

© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/3.0/).

1. Introduction

Constraint Programming is a proven technology for solving complex combinatorial problems from a range of disciplines, including scheduling (nurse rostering, resource allocation for data centres), planning (contingency planning for air traffic control, route finding for international container shipping, assigning service professionals to tasks) and design (of cryptographic S-boxes, carpet cutting to minimise waste). Constraint solving of a combinatorial problem proceeds in two phases. First, the problem is modelled as a set of decision variables with a set of constraints on those variables that a solution must satisfy. A decision variable represents a choice that must be made in order to solve the problem. Consider Sudoku as a simple example. Each cell in the 9×9 square must be filled in such a way that each row, column and 3×3 sub-square contain all distinct non-zero digits. In a constraint model of Sudoku, each cell is a decision variable with the domain $\{1 \ldots 9\}$. The

* Corresponding authors.
E-mail addresses: ian.gent@st-andrews.ac.uk (I.P. Gent), caj21@st-andrews.ac.uk (C. Jefferson), sl4@st-andrews.ac.uk (S. Linton), ijm@st-andrews.ac.uk (I. Miguel), pwn1@st-andrews.ac.uk (P. Nightingale).

http://dx.doi.org/10.1016/j.artint.2014.03.001
0004-3702/© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/3.0/).

constraints require that subsets of the decision variables corresponding to the rows, columns and sub-squares of the Sudoku grid are assigned distinct values.

The second phase is solving the modelled problem using a constraint solver. A solution is an assignment to decision variables satisfying all constraints, e.g. a valid solution to a Sudoku puzzle. A constraint solver typically works by performing a systematic search through a space of possible solutions. This space is usually vast, so search is combined with constraint *propagation*, a form of inference that allows the solver to narrow down the search space considerably. A constraint propagator is an algorithm that captures a particular pattern of such inference, for example requiring each of a collection of variables to take distinct values. A state-of-the-art constraint solver has a suite of such propagators to apply as appropriate to an input problem. In this paper we will consider propagators that establish a property called Generalised Arc Consistency (GAC) [1], which requires that every value in the domains of the variables in the scope of a particular constraint participates in at least one assignment that satisfies that constraint.

Constraint models of structured problems often contain many copies of a constraint, which differ only in their scope. English Peg Solitaire,[1] for example, is naturally modelled with a *move* constraint for each of 76 moves, at each of 31 time steps, giving 2356 copies of the constraint [2]. Efficient implementation of such a constraint is vital to solving efficiency, but choosing an implementation is often difficult.

The solver may provide a hand-optimised propagator matching the constraint. If it does not, the modeller can use a variety of algorithms which achieve GAC propagation for arbitrary constraints, for example GAC2001 [3], GAC-Schema [4], MDDC [5], STR2 [6], the Trie table constraint [7], or Regular [8]. Typically these propagators behave well when the data structure they use (whether it is a trie, multi-valued decision diagram (MDD), finite automaton, or list of tuples) is small. They all run in exponential time in the worst case, but run in polynomial time when the data structure is of polynomial size.

The algorithms we give herein generate GAC propagators for arbitrary constraints that run in time $O(nd)$ (where n is the number of variables and d is the maximum domain size), in extreme cases an exponential factor faster than any table constraint propagator [3,7,9,5,6,10–13]. As our experiments show, generated propagators can even outperform hand-optimised propagators when performing the same propagation. It can take substantial time to generate a GAC propagator, however the generation time is more than repaid on the most difficult problem instances in our experiments.

Our approach is general but in practice does not scale to large constraints as it precomputes domain deletions for all possible inputs of the propagator (i.e. all reachable subsets of the initial domains). However, it remains widely applicable — like the aforementioned Peg Solitaire model, many other constraint models contain a large number of copies of one or more small constraints.

Propagator trees

Our first approach is to generate a binary tree to store domain deletions for all reachable subdomains. The tree branches on whether a particular literal (variable, value pair) is in domain or not, and each node of the tree is labelled with a set of domain deletions. After some background in Section 2, the basic approach is described in Section 3.

We have two methods of executing the propagator trees. The first is to transform the tree into a program, compile it and link it to the constraint solver. The second is a simple virtual machine: the propagator tree is encoded as a sequence of instructions, and the constraint solver has a generic propagator that executes it. Both these methods are described in Section 3.5.

The generated trees can be very large, but this approach is made feasible for small constraints (both to generate the tree, and to transform, compile and execute it) by refinements and heuristics described in Section 4. The binary tree approach is experimentally evaluated in Section 5, demonstrating a clear speed-up on three different problem classes.

Exploiting symmetry

The second part of the paper is about exploiting symmetry. We define the symmetry of a constraint as a permutation group on the literals, such that any permutation in the group maintains the semantics of the constraint. This allows us to compress the propagator trees: any two subtrees that are symmetric are compressed into one. In some cases this replaces an exponential sized tree with a polynomially sized symmetry-reduced tree. Section 6 gives the necessary theoretical background. In that section we develop a novel algorithm for finding the canonical image of a sequence of sets under a group that acts pointwise on the sets. We believe this is a small contribution to computational group theory.

Section 7 describes how the symmetry-reduced trees are generated, and gives some bounds on their size under some symmetry groups. Executing the symmetry-reduced trees is not as simple as for the standard trees. Both the code generation and VM approaches are adapted in Section 7.3.

In Section 8 we evaluate symmetry-reduced trees compared to standard propagator trees. We show that exploiting symmetry allows propagator trees to scale to larger constraints.

[1] Problem 37 at www.csplib.org.

Download English Version:

https://daneshyari.com/en/article/6853231

Download Persian Version:

https://daneshyari.com/article/6853231

Daneshyari.com