Alternative methods for learning in CSP Solvers

Diarmuid Grimes and Richard J. Wallace
Cork Constraint Computation Centre and Department of Computer Science
University College Cork, Cork, Ireland
email: d.grimes, r.wallace@4c.ucc.ie

Abstract

This paper outlines the major features of three solvers entered in this years CPAI solver competition. The solvers shared the same underlying architecture which is described in detail in the following paper, the base solver is rjw (submitted by Richard Wallace) which Diarmuid-rndi and Diarmuid-wtdi (both submitted by Diarmuid Grimes) are built on. Furthermore the solvers all used constraint weighting as a means for identifying sources of contention in the problem. However two of the solvers (Diarmuid-rndi and Diarmuid-wtdi) used restarting to approximate sources of global difficulty, while rjw learnt local information.

1 Introduction

The solver rjw is an implementation of the classical treesearch algorithm for solving CSPs, consisting of backtracking plus lookahead (using MAC-3 for consistency maintenance). In other words, it is a complete algorithm that uses depth-first backtracking with k-way branching, interleaved with propagation. Currently it is restricted to problems involving only extensional binary constraints. This solver is designed for experimentation so the features that have received the most attention are related to this aim. Naturally efficiency is always an issue as the efficiency impacts the experiments that can be performed. The present version represents fairly mature code which has not been changed in any fundamental way over the past few years. We firstly descibe rjw then the solvers Diarmuid-rndi and Diarmuid-wtdi which build on rjw by combining learning with restarts in two quite different ways.

2 Basic Features of rjw

The solvers are implemented in Common Lisp and are designed to run on a Unix machine. Since many lisp compilers only compile into a kind of intermediate code, the program cannot run with the speed of a C or a C++ program.

The present version of rjw has a 'backbone' that is a recursive procedure (which naturally limits the size of the problems that can be handled). The basic structure is

```
search (variables, domain, solution)
    if variables == nil              /* clause 1 */
          save-solution
          return t
    else if domain == nil            /* clause 2 */
          reset data structures
          return nil                 /* backtrack */
    else if                          /* clause 3 */
          arc-consistency(next-variable, next-domain-value, remaining-variables) returns t
                and
          search (remaining-variables, new-domain, solution+next-assignment) returns t
    else                             /* clause 4 */
          return (search (remaining-variables, remaining-domain, solution))
```

Figure 1: Basic recursive structure underlying tree search in the present solver.

shown in Figure 1; as indicated, it uses recursion to run through a list of variables and a list of values in the domain of the current variable.

In the actual code, this structure is elaborated to:

- heuristically choose the next variable (in clause 3)
- set up data structures for handling arc consistency (in clause 3). (These are reset in clause 2, as indicated.)
- handle all-solutions as well as one-solution search.

In addition, the MAC solver tests for singleton domains and only does arc consistency when the current domain has more than one value. Incidentally, during search consistency maintenance is only carried out following each new instantiation (i.e. not after a value has been discarded). As per usual only arcs between a variable whose domain has changed and its unassigned neighbors are added to the queue (so the initial queue comprises solely of arcs between the variable just assigned and its unassigned neighbors). A full arc consistency is, of course, carried out prior to search.

3 Data Structures

Domain values are kept in simple lisp lists, accessed via an array. A list of variables is also used, as indicated in Figure 1. The current (partial) assignment is stored as an array, with nil values for currently unassigned variables. This array is accessed by variable-numbers, so it accomodates dynamic variable ordering. (Hence, the resetting in clause 2 in Figure 1 includes setting the value for the current variable to nil.)

In the present implementation, constraint relations are represented as arrays of binary values; the size is set by the size of the largest domain. The arrays themselves are accessed via a hash table, where the hash key is based on the two variable-numbers. Two arrays are stored for each relation, so the program does not have to put the variables in any particular order when computing the hash key. Furthermore all constraints have a weight, initially set to 1, associated with them. The weights are stored in a hashtable with a hash key based on the variable-numbers of the variables in the constraint. Whenever a constraint causes a domain wipeout during consistency maintenance the weight gets incremented by 1

The constraint representation is a global data structure that can be accessed by any function in connection with search or heuristic selection. This is also true for the current assignment. Global structures are also used to maintain a list of the original variables and the original domains.

Another important global data structure is the set of adjacency lists, which are lists of variables adjacent to a given variable in the constraint graph. Again, these are kept in an array so they can be accessed by variable-numbers. There are also data structures for certain parametric features like the degree of each variable, and the current domain sizes, and the original tightness of each constraint, which are used by certain heuristics.

A critical data structure used in connection with propagation stores current domains during search. The basic strategy (remember that this is lisp) is to maintain lists of lists within an array. Each list of lists is handled as a stack, with the current domain at the top. Using an array allows the program to access the current domain via the variable-number.

In order to use this structure during recursive search, the setup function (in clause 3 in Figure 1) adds a duplicate of the current domain to the top of each stack. (For forward checking this need only be the domains of variables adjacent to the current variable.) As successive assignments are made at a given level of search, the arc consistency functions take the domain just below the top of the stack before support testing and replace the top-most list with the adjusted domain afterwards. This means that no special (setting-up) code is required for this purpose when re-assigning a variable. If the program backtracks from a given level of search (clause 2 in Figure 1), the stacks are cut back so the domains are as they were when this level was entered.

(Incidentally, if lookahead value ordering heuristics are used [which is not done by the solvers entered in this competition], then d entries are made in these lists in the course of support checking - in the order in which values are to be assigned. This avoids any further forward checking at this level of search.)

4 Heuristics

One of the major uses to which this solver has been put in the last few years has been the experimental study of variable ordering heuristics. So there are a large number of heuristics - and anti-heuristics coded. These are organized in an elaborate but tedious manner for selecting a particular heuristic during a given run of the program.

For the competition, rjw used the domain over weighted-degree heuristic (dom/wdeg) of Boussemart et al. [**?**], in this case, the heuristic code is in the same file as the search code and the heuristic is called directly. For all solvers, values were chosen lexically (as were arcs during consistency maintenance).

5 Environment and I/O

The solvers normally run either interactively or in batch mode, where they take the same instructions from a command file. There is an i/o module that currently accepts two kinds of problem formats, both involving extensional constraints. The top-level of the program (not used in the competition) is a menu-driven system. Different top-level commands either read in the next problem (and set up most of the global data structures), or generate the next problem, or call for a solver of some generic type (e.g. backtrack, hybrid tree search, local search). During this interactive process, after an algorithm is selected, further menu options allow the heuristic to be selected for the run.

6 Restarting Strategies

The solver Diarmuid-wtdi works as follows: there is an initial failure cutoff C where search runs until C failures have occurred. It then restarts with a new cutoff which is the previous cutoff multiplied by a constant factor z. So the failure cutoff for the Rth restarted search is: $C * (z^{R-1})$. The only difference between this and the solver rjw is the restarting, so MAC-3 is the consistency algorithm and dom/wdeg is the variable ordering heuristic. The solver stops as soon as it finds a solution or proves the problem insoluble. It is complete since C is incremented with every restart.

Since weights are consistently being updated, the variable ordering is always changing, thus search is unlikely to revisit an identical part of the search space upon restarting. This allows the solver to visit different parts of the search space while still maintaining a large degree of confidence in the variables selected. In fact since at each restart it has more information available for dom/wdeg to make its early decisions it should improve its ordering with each restart. However this is contingent on the information learnt being of uniform quality which is not necessarily the case.

The solver Diarmuid-rndi is an automated learning approach to problem solving that aims to boost the power of the dom/wdeg heuristic by randomly probing the search space for information prior to a complete search using dom/wdeg. It follows Refalos third principle [**?**] which emphasises the importance of making good choices at the top of search as these have the largest impact. Refalo suggests that, in general, to achieve this one must perform some preprocessing on a problem.

The solver works as follows: similarly to Diarmuid-wtdi there is an initial failure cutoff C. However this cutoff is never incremented. Search runs identically to rjw with the one exception that variables are chosen randomly at each selection point. Constraint weights are updated throughout but are never used to guide search. If the problem is solved or the problem proven insoluble (although insolubility is unlikely to be proven since the failure cutoff is normally quite low) during these “random probes” then the solver stops.

Otherwise there is a minimum number of restarts (R_{min})which the solver will perform, at which point it will check the stability of the variable weight profile between the ${R_{min}}^{th}$ restart and the $(R_{min} - 10)^{th}$ restart. If it doesn’t satisfy some criteria for stability then the random probing continues, after every R restarts ((where R modulo

$10 = 0$) it checks the stability between the R^{th} restart and the $(R-10)^{th}$ restart until either the stability criteria has been satisfied or the number of restarts is equal to a predefined maximum number of restarts.

When either the stability criteria has been satisfied or the maximum number of restarts has been reached, search restarts for the final time. The cutoff is removed (i.e. search runs to completion), and dom/wdeg is used for variable selection with the weights learnt during preprocessing being interleaved with weights learnt during this final search.

Acknowledgment. This work was supported in part by Science Foundation Ireland under Grant 00/PI.1/C075.

References

[BHLS04] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proc. Sixteenth European Conference on Artificial Intelligence-ECAI'04*, pages 146–150, 2004.

[Ref04] P. Refalo. Impact-based search strategies for constraint programming. In M. Wallace, editor, *Principles and Practice of Constraint Programming-CP'04. LNCS No. 3258*, pages 557–571, 2004.