Mistral

1 Introduction

Mistral is a constraint library written in C++. It was created as an evolution, or rather a simplification of EFC, adapted to find *super solutions*. EFC (for Extended Forward Checking) was originally written by Fahiem Bacchus and later developed by George Katsirelos. The current version of *Mistral* was entirely rewritten, and thus it now shares very little with EFC. The initial idea was to write a light weight constraint solver, implementing *Maintain Arc Consistency* as well as the usual techniques that made the success of constraint programming (variable and value ordering heuristics, global constraint, etc). Over time, some additional features, such as restarts and branch & bound search were added. Besides a few global constraints, there is no major novelty implemented in *Mistral*.

2 Implementation

2.1 Problem Modelling

Mistral is a constraint library and not a language, hence if offers very limited modelling shortcut or syntactic sugar. Figure 1 illustrates a typical implementation of the NQUEENS problem. The types `IntVar` and `SVar` correspond respectively to a general integer variable, and to a specific implementation. A model usually involves three steps. First, the CSP and the variables are declared (lines 4 to 9). Next the constraints are posted (lines 11 to 18). Finally a solver is declared and the method `solve()` is called (lines 20 to 22), effectively solving the model.

```
 1  int main(int argc, char *argv[]) {
 2    int N = ( argc > 1 ? atoi(argv[1]) : 50 );
 3
 4    // model and variables
 5    CSP model;
 6    IntVar* queens[N];
 7    for( int i = 0; i < N; ++i )
 8      queens[i] = new SVar(N);
 9    model.add(queens, N);
10
11    // constraints
12    model.post( new AllDiffConstraint(queens, N) );
13    for(int i=0; i<N-1; ++i)
14      for(int j=i+1; j<N; ++j) {
15        IntVar *scope[2] = {queens[i], queens[j]};
16        model.post( new DiagonalConstraint(scope,
17                    queens[i]->id-queens[j]->id) );
18      }
19
20    // solver
```

```
  MACSolver s(&model, "dom/deg");
  s.solve();
}
```

Fig.1 A model for the NQueens problem in *Mistral*.

2.2 Data Structures

Four different types of variables are implemented in *Mistral*, all subclasses of `IntVar`, in other words all are finite domain integer variables.

Integer Variables as bit-vectors (SVar) In this implementation, domain $D(x)$ is represented as vector of bits. All set operations, such as union, intersection or difference can be performed in $O(n/32)$ where $n = max(D(x)) - min(D(x))$. Of course membership, insertion and deletion can be performed in constant time.

For each variable, an array of size $min(maxlevel, |D(x)|)$ of such domains is created (memory is statically allocated befor search begins). Whenever the domain change for the first time for a given level in the search tree, it is first copied in the next available index of this array. The space complexity for each variable implemented in this way is therefore:

$$min(maxlevel, |D(x)|).\frac{max(D(x)) - min(D(x))}{32}$$

Integer Variables as lists (LVar) In this implementation, a domain $D(x)$ is represented both as vector of bits mainly for checking membership, and as a doubly linked list for faster iteration. Notice that the order of the values in the list is not guaranteed to be lexicographical. This is because when backtracking, the list of deleted values is appended at the head of the domain list. Deletion of multiple values at once (such as setting the maximum or the minimum) will tend to be more expensive in this representation, however, iterating through the values of a domain is linear in the number of values.

For each variable, an array of size $min(maxlevel, |D(x)|)$ of `int` is created. Whenever the domain change at a given level, the value is removed form the domain list, appended to the head of *delta* list which is pointed by next available index in this array. When backtracking to this level, the delta list appended to the head of the domain list. The space complexity for each variable implemented in this way is therefore:

$$min(maxlevel, |D(x)|) + (max(D(x)) - min(D(x)))$$

Boolean Variables (BVar) Boolean variables have their own specific implementation for optimisation purpose. For instance, the variable does not store any data for backtracking.

Their space complexity is therefore constant (in $O(1)$).

Interval Variables (RVar) In this implementation, only a lower bound and an upper bound are stored in order to represent large intervals.

However, two array of integers (one for the lower bound and one for the upper bound) of size *maxlevel* are allocated. The space complexity of an interval variable is therefore $O(maxlevel)$.

2.3 Algorithms

Binary Backtrack Search A standard two-ways branching algorithm is implemented. It is given in integrality in Figure 2. First, the termination conditions are checked. If either the cutoff is reached (lines 2,3) or a solution is found (lines 4,5) then the status is changed accordingly and the search ends. In lines 6 to 8, the variable and value ordering heuristic is called and the next decision is selected. This decision (that is, a *left branch*) is explored in lines 10 to 18. When this branch is explored exhaustively, the complementary *right branch* is explored in line 20 to 28. Notice that when the left branch was unsuccessful because the cutoff was reached, the right branch is not explored (condition in line 22). When both branches were unsuccessful, a backtrack occurs (lines 30 to 32).

```
1     // // End of search?
2     if( limitsExpired() )
3       return (status = LIMITOUT);
4     if( allAssigned() )
5       return solutionFound();
6
7     // // Select a variable and a value
8     int idx, value = Variable::NOVAL;
9     Variable *curvar = future[(idx = heuristic->select(value))];
10
11    // // Left branch
12    curvar->makeDecision(value);
13    if( curvar->assigned )
14      bound(idx);
15    if( filtering() ) {
16      ++level;
17      if( genericBacktrack() == SAT ) return SAT;
18      --level;
19    }
20
21    // // Right branch
22    restoreTo(level-1);
23    if( status != LIMITOUT ) {
24      if( curvar->makeComplementary(value) && filtering() ) {
25        ++level;
26        if( genericBacktrack() == SAT) return SAT;
27      } else ++level;
28      // // Backtrack
29      restore();
30      ++BACKTRACKS;
31    } else ++level;
32
33    return status;
```

Fig.2 *Mistral*'s backtracking procedure.

Generic Arc Consistency Algorithm The *arc consistency* algorithm used for extensionally defined constraints is a slightly modified version of the simple `AC3` algorithm. This modification is sometime called *residual* `AC3` [7]. Whenever a support is found for a given pair ⟨ variable, value ⟩, it is stored. Then the next time a support need to be found for this pair, the stored support is checked for validity (i.e., whether the values involved in this support are still in their respective domains). If not, the regular `AC3` algorithms proceeds as normal. Since nothing needs to be done when backtracking (as opposed to `AC2001` for instance), the overhead in time complexity is marginal, whilst the gain is in practice noticeable.

3 Solver Competition

3.1 Representations of Variables

The different implementations of variables were used according to heuristic rules. The simplest one being that Boolean Variables were always represented using `BVar`. The range variables (`RVar`) were only used when extra variables with very large domains (larger than "*maxlevel*") were required. This is explained in more detail in Section 3.2. Finally the choice between `SVar` and `LVar` was down to the size and the density of the domain. The list implementation `LVar` is intuitively better when the ratio $\frac{D(x)}{max(D(x))-min(D(x))}$ is low, since being able to iterate in linear time through a sparse domain is valuable. Moreover, when the size of the domain increase, the space used for the list is outbalanced by the conciseness of the backtracking structure (a single integer as opposed to a domain for `SVar`). Therefore `LVar` was used for large and/or sparse domains, whilst `SVar` was used in all other cases.

3.2 Representations of Constraints

Relations Relations are represented as an n dimensional matrix flattened into a vector of bits. The worst case space complexity is not very good, since sparse or dense matrix have the same size. Moreover, in order to keep the membership operation in constant time, the size of a binary relation between x and y is stored using $(max(D(x)) - min(D(x))).(max(D(y)) - min(D(y)))$ bits, instead of $|D(x)|.|D(y)|$. However this was never a problem during the first round of the competition.

Predicates Two different representations of predicates were used, both of them pretty standard. During the first round of the competition only the first version was implemented, and comported several bugs. Given a tree of binary and unary predicates, a set of as many respectively ternary and binary reified constraints and extra variables are created. For instance for the predicate:

$$eq(add(mul(X_0, X_1), X_2), X_3)$$

the following constraints will be posted:

$$\begin{aligned} mul(X_0, X_1, Y_0) \\ add(Y_0, X_2, Y_1) \\ eq(Y_1, X_3) \end{aligned}$$

where Y_0 and Y_1 are extra variables; $mul(X_0, X_1, Y_0)$ constrains the product of X_0 and X_1 to be equal to Y_0; $add(Y_0, X_2, Y_1)$ constrains the sum of Y_0 and X_2 to be equal to Y_1; and $eq(Y_1, X_3)$ will substitute X_3 to Y_1 in all other constraints. Notice that the latter substitution is only possible because the constraint eq is at the root of the predicate tree, otherwise a ternary constraint $eq(Y_1, X_3, Y_2)$ would be posted, constraining Y_2 to be the truth value of the relation $Y_1 = X_3$.

The second representation is used for low arity constraints encoded as a predicates tree. In this case instead of extra variables and constraints, a unique constraint is posted. This constraint is propagated using the generic arc consistency algorithm, that is, the algorithm used for extensional constraints. However, instead of implementing constraint checks using a Boolean matrix, the predicate tree is stored and queried at each constraint check. This is essentially equivalent to transforming the predicate into a table constraint, albeit with slightly worse time complexity and better space complexity.

3.3 Search Strategy

No specific value heuristic was used in the competition, the values are therefore visited in lexicographical order for `SVar`, `RVar` and `BVar` and in an undefined order for `LVar`. The variable ordering heuristic is a slightly modified version of *domain over weighted degree* [4]. As in the regular framework, each constraint $C(V)$ over a set of variable V is associated with a weight $w(C)$ and the variable with minimum ratio *domain size* over *sum of neighbouring constraints weights* is chosen:

$$\text{choose } x \text{ such that } \frac{|D(x)|}{\sum_{x \in V} w(C(V))} \text{ is minimum}$$

However, on failure during the GAC closure procedure, the constraint responsible for the failure gets its weight incremented by $maxlevel - level$ instead of 1. The intuition behind this choice is that a failure early in the search is more meaningful than a failure later, since less decisions have been taken.

Two versions of *Mistral* were submitted in the competition, one of them implementing a geometric restart policy. The initial cutoff was set to $\frac{2}{3}.maxlevel$, and then multiplied by $1 + \frac{1}{3}$ upon every restart. It is worth noticing that a very limited form of nogood learning naturally happens when restarting the basic backtracking procedure illustrated in Figure 2. Indeed, the procedure is called with the level set to 1. When a left branch, for instance the decision $X = v$, is unsuccessful on level 1, then the complementary decision $x \neq 1$ is explored. However, this decision is globally consistent and therefore never withdrawn, therefore $x \neq 1$ is a unary nogood, reused upon subsequent restarts. Moreover, observe that arbitrarily many left branches and many variables may be chosen at level 1. Therefore, several unary nogood, involving different variables may be “learnt”.

4 Features

Some novel global constraint propagation algorithm have been implemented in *Mistral* in order to perform empirical evaluation. The NVALUE constraint, for example, as described in [2] is implemented in *Mistral*.

The NValue constraint counts the number of distinct values used by a vector of variables. It is a generalisation of the widely used AllDifferent constraint [5, 8]. It was introduced in [6] to model a musical play-list configuration problem so that play-lists were either homogeneous (used few values) or diverse (used many). There are many other situations where the number of values used are limited. For example, if values represent resources, we may have a limit on the number of values used at the same time. A NValue constraint can thus aid both modelling and solving many real world problems.

Definition 1. NValue$(N, [X_1, \ldots, X_m])$ *holds iff* $N = |\{X_i| \ 1 \leq i \leq m\}|$

Enforcing generalised arc consistency (GAC) on the NValue constraint is NP-hard [3]. One way to deal with this intractability is to decompose the constraint or to approximate the pruning. The NValue constraint can be decomposed into two other global constraints: the AtMostNValue and the AtLeast-NValue constraints. Unfortunately, while enforcing GAC on the AtLeast-NValue constraint is polynomial, enforcing GAC on the AtMostNValue constraint is also NP-hard. We will therefore focus on various approximation methods for propagating the AtMostNValue constraint.

Mistral features three new approximations. Two are based on graph theory while the third exploits a linear relaxation encoding. These algorithms compare favourably to a previous approximation method due to Beldiceanu based on intervals that runs in $O(n \log(n))$ [1]. The two new algorithms based on graph theory are incomparable with Beldiceanu's, though one is strictly tighter than the other. Both algorithms, however, have an $O(n^2)$ time complexity. However, the linear relaxation method dominates all other approaches in terms of the filtering, but with a higher computational cost.

References

1. N. Beldiceanu. Pruning for the *Minimum* Constraint Family and for the *Number of Distinct Values* Constraint Family. In Toby Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 211–224, Paphos, Cyprus, 2001. Springer-Verlag.
2. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Filtering Algorithms for the NValue Constraint. In Roman Barták and Michela Milano, editors, *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-05)*, volume 3524 of *Lecture Notes in Computer Science*, pages 79–93, Prague, Czech Republic, 2005. Springer-Verlag.
3. C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. Tractability of Global Constraints. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 716–720, Toronto, Canada, 2004. Springer-Verlag. Short paper.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting Systematic Search by Weighting Constraints. In Ramon López de Mntaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 482–486, Valencia, Spain, August 2004. IOS Press.

5. M. Dincbas, P. van Hentenryck, H. Simonis, and A. Aggoun. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, 1988.
6. P. Roy F. Pachet. Automatic Generation of Music Programs. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 331–345, Alexandria, VA, USA, 1999. Springer-Verlag.
7. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc Consistency in MAC: a new perspective. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP-04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 93–107, Toronto, Canada, October 2004. Springer-Verlag.
8. J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.