An Interval Calculus Based Finite Domain Constraint Solver and Its Implementation In Prolog

Jin-Kao HAO and Jean-Jacques CHABRIER
Centre de Recherche en Informatique de Dijon *
Faculté des Sciences Mirande - B.P. 138 - 21004 Dijon Cédex - France
E-mail: alf@frccub11.bitnet

Abstract: Constraint logic programming (CLP) is a powerful paradigm combining constraint and logic programming. At the heart of any CLP system there is a constraint solving mechanism i.e. constraint solving algorithm. In this paper, we present the algorithm and its Prolog implementation of a finite domain CLP system called Conslog. The algorithm is a variation of Waltz-like algorithm combining backtracking. It uses the interval calculus to largely reduce the search space, a dynamic constraint selection rule to choose constraints to be refined and a heuristic variable selection rule for value generations. Particular attention is devoted to detail the interval calculus as constraint refining technique. An example is presented. Some constraint solving algorithms such as generate-and-test, backtracking, forward-checking and consistency-checking are reviewed and discussed.

1. Introduction

Constraint Logic Programming (CLP) [Jaffar & Lassez 87] is a powerful programming paradigm combining constraint programming and logic programming. Constraint programming aims at solving Constraint Satisfaction Problems (CSPs) by using some efficient constraint solving techniques. Logic programming, thanks to its declaritivity and its expression power, can be used to elegantly formulate many CSPs as well as other applications. However, solving CSPs in Prolog-like systems is by no means efficient. The brute-force "generate-and-test" solving strategy of these systems is too primitive. CLP tries to put together the best features of these two paradigms. Several CLP have been reported, for example, Prolog III [Colmerauer 87], CLP(R) [Jaffar & Michaylov 87], and Chip [Dincbas et al 88].

Recently, a new approach to CLP has been proposed [Hao & Chabrier 90a]. Philosophically, grouping a set of functionalities into a single unit often causes problems such as overheads, difficulty of implementation and unextensibility. Specializing the functionalities into different parts is more desirable. Contrary to the previous systems which are built in some "tightly coupled" manner, the proposed approach distinguishes a pre-compilation phase and a proper constraint solving phase. The last can be probably further decomposed into static and dynamic solving of constraints. This approach favors optimizations by avoiding some useless computation. Also, systems based on this approach have the desirable modularity. Implementations become easier because different functionalities are isolated into different parts. A prototype called Conslog, which is based on the proposed approach, has been implemented in Prolog. The Conslog system consists of two parts, a partial evaluator [Hao & Chabrier 90b] and a constraint solver in finite domain. The partial evaluator is used to pre-compile constraint logic programs. The pre-compilation consists mainly of carrying out logical inferences, unfolding predicate callings, propagating instantiated values and collecting constraints. The collected constraints constitute the input for the constraint solver. *The partial evaluator establishes a kind of "bridge" between logic programming and constraint solving.* Thanks to the partial evaluator, the problem remained for the solver becomes a pure constraint solving problem. Many techniques developed in constraint programming can be investigated and integrated. Besides, the system is extensible. Vertically, each part of the system can be improved independently. Horizontally, other solvers for other domains can be built and connected to the partial evaluator without affecting the existing system.

In this paper, we will concentrate on algorithmic aspects of Conslog's actual solver and its implementation issues in Prolog. The solver uses a variation of Waltz-like algorithm combining backtracking. The algorithm integrates the interval calculus to largely reduce domains of variables of constraints. Forward-checking is equally used to deal with disequality constraints. Special attention will be put on the refining techniques based on interval calculus.

2. Problem Definition: Finite Domain CSPs

A finite domain CSP is defined by a finite set of variables, a collection of finite sets of possible values and a finite set of n-ary relations on variables. These relations are called constraints. We will

---

* The work is partially supported by the ALF project (Esprit N° 1520)

consider a special class of CSPs where variables take their values from positive integers. Many problems ranging from puzzles to integer programming such as cryptarithmetic problem, map colouring, resource allocation, scheduling and so on fall into this class. A CSP can be represented more formally by a 3-tuple {V,D,C} with

$V=\{ V_1,V_2,...,V_n \}$ set of variables

$D=\{ D_1,D_2,...,D_n \}$ collection of possible value sets

where $D_i=\{ d_{ij} \mid d_{ij}$ is a possible value of $V_i \}$ $(1\le i\le n)$ called domains of variables

$C=\{ C_i(V_{i1},...,V_{ik}) \mid C_i$ is a relation on variables $V_{i1},...,V_{ik}\in V \}$ $(1\le k\le n)$

Solving a CSP means to determine all possible assignments of values to variables such that all the constraints are satisfied at the same time. A value $d_{ij}\in D_i$ of Vi is consistent with respect to constraint $C_i(X,Y,...,Vi,...Z)$ if $\exists d_1\in D_X$, $d2\in D_Y$,...., $dp\in D_Z$ such that $C_i(d_1,d_2,...,d_{ij},...,dp)$ is true. Otherwise d_{ij} is inconsistent. A variable V_i is consistent if $\forall d_{ij}\in D_i$, d_{ij} is consistent with respect to the constraint Ci(...,Vi,...). A constraint $Ci(V_{i1},...,V_{ik})$ is consistent if V_{ij} j=1,...,k are consistent. Note that reaching consistency for all constraints involved in a CSP is only a neccessary, not a sufficient condition for solving the CSP.

3. Algorithms for CSPs

There exist several algorithms for CSP solving [Mackworth 87]. Mackworth classifies them into two categories: backtracking and consistency-checking. The simplest algorithm consists of generating a value $d_{ij}\in D_i$ for each $Vi\in V$ involved in at least one $Ci\in C$ and then testing the truth values of all the $Ci\in C$. If the conjunction of these Ci is evaluated to false, another assignment is generated and tested again. This "generate-and-test" algorithm is correct but very inefficient, because the generating space $D=D_1*D_2*...*D_n$ may be very large. A first improvement concerns generating partial assignment *in some fixed order*. As soon as some $Ci\in C$ becomes instantiated, its truth value is tested soon. If Ci is false, the actual partial assignment can not be part of any final solution. Backtracking occurs on the last instantiated variable with untried values. The advantage of this backtracking algorithm is that the search space below the failure is cut down. However, backtracking suffers from the "thrashing" behavior [Mackworth 77] which characterizes the fact that the backtracking doesn't remember already discovered failures and may rediscover many times the same failures in the later backtrack search. Note that solving mechanisms of Prolog-like systems implement this kind of searching. However, since many CSPs are NP-complete, there is no hope to solve these problems in polynomial time in general. Backtracking algorithms are unavoidable. The best one can do is to incorporate some techniques such as intelligent backtracking and consistency-enforcing to reduce the most the later search space.

One of such techniques is forward-checking [Haralick & Shapiro 80]. In forward-checking algorithms, we generate also values for variables. However, after each generation, Constraints are used to logically delete inconsistent values of other variables making the constraints consistent. Then we instantiate another variable, eliminate all the inconsistent values and so on. At each stage, the remaining values of future variables are consistent with past instantiations with respect to the constraints. Search space is definitely reduced each time an inconsistent value is eliminated. It is clear that reasonning with forward-checking is different from the "generate-and-test". By comparison with the passive use of constraints in "generate-and-test", using constraints in this way is called active use of constraints [Gallaire 85]. Forward-checking in logic programming has been first defined and investigated in [Van Hentenryck and Dincbas 87]. A constraint is said forward checkable if all but one variable of the constraint are instantiated. Disequality ($\neq$) constraints are best suited for the forward-checking. For other constraints containing N-variables (N>2) other techniques such as interval calculus are needed. For example, with the above definition of forward-checking, constraint X+2*Y+Z=V+3*W can not be activated, while interval calculus can probably reduce the domains of variables directly from the constraint. We will develop this idea in the next section.

Another family of algorithms concerns so-called consistency algorithms [Mackworth 77]. The main idea is similar to that of forward-checking, i.e. constraints are actively used to eliminate inconsistent values that can not participate in any final solution. Deleting inconsistent values from domains by using constraints is called consistency-checking or constraint refining. Like forward-checking, the deletion of one inconsistent value may lead to deletion of another, which may lead to further deletion of another, and so on. The successive deletion of inconsistent values constitutes "constraint propagation". The difference of this class of algorithms with the previous ones is that here, we don't generate values for variables. A variable V_i gets its value d_{ij} only when its domain D_i is reduced to a singleton $\{d_{ij}\}$. If, during the consistency-checking, some variable's domain is reduced

to empty, there exists some contradiction among the constraints. Consistency can be reached at several levels in terms of constraint networks. There exist algorithms for reaching node, arc, path and k-consistency (k ≤ the number of variables in a given CSP) [Freuder 78]. Waltz's filtering [Waltz 74] is an arc consistency algorithm. Whether a consistency algorithm can find solutions depends on the consistency level that can be reached by the algorithm. In general, consistency algorithms alone can not solve CSPs completely. Some variables may still have multiple values in their domains after the application of algorithms. Instantiating these variables requests some search techniques. Algorithms integrating consistency checking and searching are called cooperative algorithms. We present now the algorithm of Conslog's solver which is a variation of Waltz-like algorithm combining backtracking.

4. The Algorithm of Conslog's Solver

4.1. General Algorithm

The proposed approach in [Hao & Chabrier 90a] makes the work of the solver simple. The solver works with a set of constraints together with domain information which come from the partial evaluator. Constraints take the form $\Sigma Ci*Xi \odot \Sigma Dj*Yj$ with Ci, Dj≥0, Xi, Yj variables of positive integer and $\odot \in \{=, \neq, >, \geq, <, \leq\}$. A constraint is called active if domains of its variables can be possibly reduced by this constraint. Note that this definition is less restricted than that for the forward-checkable mentioned in the previous section. Changes in a variable's domain make some constraints active (woken constraints), but not necessarily forward-checkable. To refine a variable by a constraint is to reduce the variable's domain by deleting inconsistent values with respect to the constraint. To refine an (active) constraint means to use the constraint to refine all its variables. The refining of variables and constraints is carried out by refining procedures which will be defined later. A constraint is solved if it is sufficiently instantiated and its truth value is evaluated to true.

```
      Algorithm:
      ACTIVE <-- Set of active constraints; NON-ACTIVE <-- Set of non active constraints;
      WHILE ACTIVE ≠ Empty & NON-ACTIVE ≠ Empty DO
          BEGIN
S1:         WHILE ACTIVE ≠ Empty DO
                  BEGIN
                  a: Choose a constraint C from ACTIVE;
                  b: Refine C   /* if the domain of a variable is reduced to empty,
                                   there is some inconsistency. Backtrack or stop with failure */
                  c: Move woken constraints from NON-ACTIVE to ACTIVE;
                  d: IF C is solved THEN Throw C away ELSE Put C into NON-ACTIVE;
                  END { while}
S2:               a: Choose a variable X;
                  b: Generate a value belonging to its domain Dx;
                  c: Move woken constraints from NON-ACTIVE to ACTIVE;
          END {while}
S3:   IF ACTIVE = Empty & NON-ACTIVE = Empty THEN End with success
```

In the algorithm, S1.a, called constraint selection rule (CSR), corresponds to the computation rule in logic programming. In our system, a dynamic CSR is used. The CSR chooses first the strongest constraint to refine. For example, E≠7 will be chosen before E+B+2*C=5*A+D. S1.b contains different refining procedures. Consistency-checking happens here. Inconsistent values of variables with respect to the current context will be eliminated by the refining procedures. A constraint after refining will become consistent in the sense of §2. For different kinds of constraints, there are different refining procedures. For ≠ constraints, forward-checking is used. For =, <, ≤, >, ≥ constraints, interval calculus is used. If a domain after refinement (RD) becomes a singleton, the single value is directly assigned to the variable and propagated to other constraints. If RD is empty, backtracking occurs on generated values by S2. If the constraint is completely instantiated, its truth value is tested. A false value leads also to backtracking. S2 is a generating procedure defining a heuristic variable selection rule (VSR). When there is no more active constraint and there are still unsolved constraints, the VSR selects the smallest domain variable and generates for it a value from its domain. The generated value is propagated to all concerned constraints. This makes some of constraints active and thus stimulates again the above refining procedures. S3 corresponds to the successful ending of the algorithm when all the constraints are solved.

We see that a variable can get its values in two ways. One concerns the generated values in S2.b. The other is that variable's domain is reduced into a singleton by the refining procedures in S1.b. These two ways of instantiating variables are very different in nature. The first consists of arbitrarily giving a value and then waiting to see if this instantiation is consistent. On the contrary, an instantiation in the second way is a logical consequence implied by the constraints.

4.2. Interval Calculus Based Refining Procedure

General interval arithmetic has been studied in [Alefeld & Herzberger 83], discussed in [Davis 87]. We use the term "Interval Calculus" to denote the limited interval arithmetic in integer. The interval calculus has been discussed and used in Alice [Laurière 76] and Chip [Dincbas et al 87].

An interval is defined as a set of integers in increasing order. An interval is consecutive if every integer between its two bounds belongs to the interval. Otherwise it is non-consecutive one. A consecutive interval can be represented by its two bounds. For a non consecutive interval, it is represented by enumerating all its elements between lower and upper bounds. First we suppose that all intervals are consecutive. Let's look at a very simple example. We have ten coins of one dollar, ten coins of two dollars and ten coins of five dollars, we want to know how many ways there may be to get 21 dollars. The problem can be easily defined by the constraint $X+2*Y+5*Z=21$ with $X,Y,Z\in\{1,...,10\}$. Before beginning any enumeration, we use the constraint to refine the constraint. Since $2*Y=21-(X+5*Z)\leq 21-(1+5*1)=15$, we get $Y\leq 7$. Similarly, we have $Z\leq 3$ and X is unchanged. Reasonning like this constitutes the basic idea of the interval calculus based refining procedures. General constraints of the form $P\leq\sum Ci*Xi\leq Q$ with Ci, P, $Q\geq 0$, $Xi\in\{ai,bi\}$ can be refined using the following formula:

$$Xi\in\{\max\{ai,(P-B)/Ci\},\ \min\{bi,(Q-A)/Ci\}\}\quad \text{with } A=\sum Cj*aj,\ B=\sum Cj*bj\ (j\neq i)$$

Constraints of the form $\sum Ci*Xi \copyright \sum Dj*Yj$ with Ci, $Dj\geq 0$, Xi, Yj variables and $\copyright\in\{=, >, \geq, <, \leq\}$ can be effectively refined after transforming into the above form. In the following, we detail the refining procedure for constraints of the form $\sum Ci*Xi=\sum Dj*Yj$ with $Xi\in\{ai,bi\}$, $Yj\in\{uj,vj\}$. For other constraints, refining proceduces can be defined similarly. The refining steps to be taken for $\sum Ci*Xi=\sum Dj*Yj$ are:

step 1: Calculate $A1=\sum Ci*ai$, $B1=\sum Ci*bi$, $A2=\sum dj*uj$, $B2=\sum dj*vj$

step 2: $A=\max\{A1,A2\}$, $B=\min\{B1,B2\}$

step 3: If $\neg(A>B)$ then refine $A\leq\sum Ci*Xi\leq B$ and $A\leq\sum Dj*Yj\leq B$ else there is a contradiction

Or better, we can distinguish different cases for step 3.

case 1: A=B then refine $A\leq\sum Ci*Xi\leq A$ and $A\leq\sum Dj*Yj\leq A$

case 2: A<B then

if A1<A2(so A=A2) then

- B1<B2(so B=B2) then refine $A\leq\sum Ci*Xi$ and $\sum Dj*Yj\leq B$
- B1>B2(so B=B1) then refine $A\leq\sum Ci*Xi\leq B$
- B1=B2 then refine $A\leq\sum Ci*Xi$

if A1>A2(so A=A1) then

- B1>B2(so B=B2) then refine $\sum Ci*Xi\leq B$ and $A\leq\sum Dj*Yj$
- B1<B2(so B=B1) then refine $A\leq\sum Dj*Yj\leq B$
- B1=B2 then refine $A\leq\sum Dj*Yj$

if A1=A2 then

- B1>B2(so B=B2) then refine $\sum Ci*Xi\leq B$
- B1<B2(so B=B1) then refine $\sum Dj*Yj\leq B$
- B1=B2 then $\sum Ci*Xi=\sum Dj*Yj$ is consistent, no refinement is needed

case 3: All other cases are contradiction

It is natural to do refining for consecutive intervals, because the interval calculus is based only on bounds of intervals. The technique can be used to deal with non consecutive intervals as well. The domain after refinement (RD) is not a simple replacement by the resulting interval (RI) from interval calculus, rather it is the intersection between the initial domain (ID) and the resulting interval, i.e. $RD=\{vi \mid vi\in ID \ \&\ vi\in RI\}$. Thus the existing "holes" are kept in the resulting domain. For binary constraints, the interval calculus is simplified to the comparison of different bounds of two domains. For $\neq$ constraints $\sum Ci*Xi\neq\sum Dj*Yj$, interval calculus can not help much and forward-checking technique is used. For example, from $X\neq Y$ with $X\in\{2,3,4\}$ and $Y\in\{2,3,4\}$ nothing new can be deduced. With this kind of constraints, we can not do better than forward-checking, i.e. we must wait until all but one variable is left uninstantiated.

5. A Prolog Implementation of the Algorithm

The algorithm and the different refining techniques presented above have been implemented in Sepia Prolog [Meier et al 88]. The input of the solver consists of a goal to be solved, a set of constraints and the variable-domain information. Only the goal is explicitely given to the solver, the rest of the input data comes from a file created and used by the partial evaluator. Data structures used are mainly lists. A set of constraints is represented by a list. Variable-domain information is represented by a list of pair (X,Dx). The domain Dx itself is a list containing the possible values of the variable X. The implementation consists of essentially three parts: a control structure, refining procedures and a value generation procedure. The control structure corresponds to the two WHILE_DO loops in the algorithm. It is a four parameters predicate control (Goal, Actives, Non_Actives, Var_Dom) which says "Given a set of constraints "Actives"+"Non_Actives" with domain information "Var_Dom", find all variable bindings for the variables in "Goal" ". The refining procedures are based on interval calculus and forward-checking. Encoding them in Prolog is technical but almost straightforward. In total, the solver requires about nine hundreds lines of code. No special language features such as delay or corouting are needed. More details can be found in [Hao & Chabrier 89].

6. An Example

Find for each letter in the formula SEND+MORE=MONEY a different number between 0 and 9 such that the formula holds in the decimal system. The problem is easily stated in Conslog as follows:

```
domain(sendmory([0,..,9],[0,..,9],[0,..,9],[0,..,9],[0,..,9],[0,..,9],[0,..,9],[0,..,9]).
sendmory([S,E,N,D,M,O,R,Y]):-
        M≠0, S≠0, 1000*S+91*E+10*R+D = 9000*M+900*O+90*N+Y,
        alldifferent([S,E,N,D,M,O,R,Y]).
alldiff([X|Xs]):- outof(X,Xs), alldiff(Xs).
alldiff([ ]).
outof(X,[Y|Ys]):- X≠Y, outof(Ys).
outof(X,[ ]).
```

The program is first pre-compiled (once and for all) by the partial evaluator with the goal?:- peval(sendmory([S,E,N,D,M,O,R,Y])). The partial evaluator **peval** collects the first three constraints of "sendmory". When it reaches "alldiff", unfolding takes place. Unfolding "outof" makes explicit all ≠ constraints. Finally, the partial evaluation produces a file containing the constraints: {M≠0, S≠0, 1000*S+91*E+10*R+D=9000*M+900*O+90*N+Y, S≠E,S≠N,S≠D,S≠M,S≠O, S≠R, S≠Y, E≠N,E≠D, E≠M, E≠O, E≠R, E≠Y, N≠D, N≠M, N≠O, N≠R, N≠Y, D≠M, D≠O, D≠R, D≠Y, M≠O, M≠R, M≠Y, O≠R, O≠Y, R≠Y}. Note that, thanks to the dynamic CSR, the order of the constraints is not important at all. Now the solver can be run with the goal ?:-conslog(sendmory([S,E,N,D,M,O,R,Y])). M≠0 and S≠0 are successively chosen and refined removing 0 from D_M and D_S. Next the only equality constraint is refined by the interval calculus based refining procedure, giving S to 9, M to 1, and reducing D_O to {0,1}. Refining ≠ constraints leads to the removal of 9 and 1 from concerned domains instantiating O to its single possible value 0. The equality constraint, which becomes active again, is refined giving the following state: S=9, M=1, O=0, E∈{4,...,7}, N∈{5,...,8}, D,R,Y∈{2,...,8}. Now there is no more active constraint. The VSR is called to choose the smallest domain variable E. Instantiating E to 4∈D_E leads to a contradiction. Backtracking occurs on next untried value 5∈D_E. Refining the equality constraint instantiates N to 6 and R to 8 and reduces D_D to {7,8} and D_Y to {2,3}. Removing 8∈D_D by D≠R instantiates D to its single possible value 7. Refining again the equality constraint instantiates the last variable Y with 2. In total, we obtain the result in less than 2 seconds on a sun3/110 with only one backtrack. Note that the solver works at meta level.

Other problems such as N-queens, zebra, magic square and map colouring problems can be also specified and solved easily.

7. Related Work

Generally speaking, our work is related to CLP systems such as Prolog III, CLP(R) and especially Chip. However, there are some differences. The most important difference is the philosophical one discussed in the first section. Another difference with Prolog-III and CLP(R) is that they work in numerical domains such as rational and real with the Simplex algorithm. Our work has been influenced by some constraint manipulation ideas of the Alice system. Alice supports some kind of mathematical language. The language offered by Conslog takes the form of logic programming. At the technical level, unlike Alice, Conslog doesn't create new constraints during constraint solving.

8. Conclusion

The algorithm and its implementation issues of a finite domain constraint solver are presented. The solver is an independent part of a finite domain CLP system Conslog. The solver integrates constraint refining procedures into a Waltz-like algorithm combining backtracking. Particular attention has been put on the using of the interval calculus as constraint refining technique. Forward-checking is also used to deal with $\neq$ constraints. The CSR and VSR make almost unimportant the constraint order in a program. An example is described to show the behavior of the system. The implementation in Prolog was almost straightforward and caused no special difficulties. Our experiences showed that new ideas about CLP can be easily prototyped with existing logic programming systems. An experimental laboratory has been built to do further investigations. Prototypes built as such are portable. The techniques described here could be aslo useful for real-valued problems. Besides, the solver can be used to solve boolean problems where variables takes values from {0,1}.

Acknowledgements

The authors wish to thank the referees of the paper for helpful comments and suggestions.

Bibliography

[Alefeld & Herzberger 83] G. Alefeld and J. Herzberger, Introduction to Interval Computation. Chapter1 and 2, Academic Press, New York, 1983.

[Colmerauer 87] A. Colmerauer, Opening the Prolog III universe, BYTE, August 1987.

[Davis 87] E. Davis, Constraint propagation with interval labels. J. of AI, Vol. 32, pp281-331, 1987.

[Dincbas et al 87] M. Dincbas, H, Simonis and P. Van Hentenryck, Extending equation solving and constraint handling in logic programming. In MCC (Ed;) Colloquium on Resolution of Equations in Algebraic Structures, Texas, USA, May 1987.

[Dincbas et al 88] M. Dincbas, P. Van Hentenryck and H, Simonis, A. Aggoun, T. Graf and F. Berthier, The constraint logic programming language CHIP. In Proc. of the Int. Conf. on Fifth Generation Computer Systems 1988, pp693-702, ICOT, Tokyo, Japan, 1988.

[Freuder 78] E.C. Freuder, Synthesizing constraint expressions. Communication of ACM, Vol. 21, pp958-966, November 1978.

[Gallaire 85] H. Gallaire, Logic programming: future developments. In Proc. IEEE Symposium on Logic programming. Invited paper, pp88-96, Boston, 1985.

[Hao & Chabrier 89] J.K. Hao and J.J. Chabrier, An interval calculus based finite domain constraint solver and its implementation in Prolog. TR89-H-2, CRID, 1989.

[Hao & Chabrier 90a] J.K. Hao and J.J. Chabrier, Combining partial evaluation and constraint solving: a new approach to constraint logic programming. In Proc. of SPLT-90, pp431-446, Trégastel, France, May 1990.

[Hao & Chabrier 90b] J.K. Hao and J.J. Chabrier, A partial evaluator and its application to constraint logic programming. TR90-H-1, CRID, 1990.

[Haralick & Shapiro 80] R.M. Haralick and L.G. Shapiro, Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, Vol.14, pp263-313, 1980.

[Jaffar & Lassez 87] J. Jaffar and J-L. Lassez, Constraint logic programming. In Proc. of POPL-87, Munich, Aug. 1987.

[Jaffar & Michaylov 87] J. Jaffar and S. Michaylov, Methodology and implementation of a CLP system. In Proc. of ICLP-4, Melbourne, May 1987.

[Laurière 76] J-L. Laurière, Un langage et un programme pour énoncer et résoudre des problèmes combinatoires. Thesis, Université de Paris VI, May 1976.

[Mackworth 77] A.K. Mackworth, Consistency in networks of relations. Artificial Intelligence. Vol.8, pp99-118, 1977.

[Mackworth 87] A.K. Mackworth, Consistency satisfaction. In S.C. Shapiro (Ed.) Encyclopedia of Artificial Intelligence, John Wiley & Son, New York, 1987.

[Meier et al 88] M. Meier, G. Macartney, P.A. Tsahaheas, D.H. De Villeneuve and D. Chan, Sepia User Manual. TR-LP-38, ECRC, September 1988.

[Van Hentenryck & Dincbas 87] P. Van Hentenryck and M. Dincbas, Forward-checking in logic programming. In Proc. of 4th ICLP, pp229-256, Melbourne, Australia, May 1987.

[Waltz 75] D. Waltz, Understanding line drawings of scenes with shadows. In P.H. Winston (Ed.), The psychology of Computer Vision, McGraw-Hill, New York, 1975.