A Constraint-Space Based CSP Solving Method

Gheorghe Ghiculete and Stefan Trausan-Matu

Expert Systems Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA

Abstract: A new method for solving constraint satisfaction problems over variables with finite domains is presented. This method focuses on the problem's **constraints** rather than on its **variables** - as most of the 'classic' approaches do. Some advantages and disadvantages of the method are presented, as well as some preliminary test results.

1. Introduction

After more than twelve years of studies, constraint programming is today an important branch of Artificial Intelligence. A *constraint* is a mathematical relation over an (usually small) set of variables. Constraints may be connected (via variables) to forming *constraint networks*. Constraint networks may be classified using several criteria: the domain of the variables (discrete, continuous, intervals, multiple-values, a.s.o.), the type of the relations involved (e.g. algebraic, rules), the techniques used to solve the net (value propagation, symbolic propagation, relaxation schemes, filtering), the way constraints may be added/removed (static, dynamic). With respect to the variable domain type, an important category of constraint networks is the CSP class. CSP stands for Constraint Satisfaction Problem - problems over variables with finite value domains. This paper focuses on this kind of constraint nets by presenting the CSP module embedded in the COPE general constraint representation and processing system. The CSP description is followed by an overview of the existing solving methods. In the next section a new approach to CSP solving is presented, together with some tests and their results, which are quite encouraging. A sample problem is listed in the Appendix.

2. Constraint Satisfaction Problems

A CSP may be described as 3-tuple (X,D,F) where $X=\{x_1...x_n\}$ is a set of variables, $D=\{d_1...d_n\}$ the associated domains, and $F=\{f_1...f_m\}$ a set of Boolean functions called 'constraints'. Domains are characterized by their cardinality $card_i$. Each constraint fi may be regarded as a predicate over a subset of X. We shall refer the constraint arity of fi as k_i. To keep things less complicated let us suppose that $card_1 = ... = card_n = d$ and $k_1 = ... = k_n = k$. Finding a solution to a CSP means searching for an assignation $\{x_1=v_1,...,x_n=v_n\}$, where v_1 in d_1,...,v_n in d_n, so that all the constraints should be satisfied (i.e. $f_1...f_m$ all return TRUE for that assignation).

For example, let $X=\{x_1,x_2,x_3\}$, $D=\{d_1,d_2,d_3\}$, $d_1=\{a,m,n\}$ $d_2=\{b,p,q\}$ $d_3=\{c,x,y\}$ and $F=\{f_1,f_2,f_3\}$, the constraints being defined as
$f_1(x_1,x_2) = (x_1 = a \text{ and } x_2 = b \text{ or } x_1 = m \text{ and } x_2 = b)$
$f_2(x_2,x_3) = (x_2 = b \text{ and } x_3 = c \text{ or } x_2 = b \text{ and } x_3 = x)$
$f_3(x_1,x_3) = (x_1 = a \text{ and } x_3 = c \text{ or } x_1 = n \text{ and } x_3 = x)$. It is easy to realize that the only assignation that satisfies this simple example is $(x_1=a, x_2=b, x_3=c)$.

A brief description of the 'classic' method for solving a CSP is the following: 1. (optional) impose an instantiation order on the variables set 2. (forward step) instantiate the variables one at a time; check, after each instantiation, the applicable constraints, that is, every $f(x_1,...,x_k)$ with $x_1,...,x_k$ all assigned. If at any point (partial assignment) a constraint is violated then a backward step is performed: one of the previous

variables in the ordering is once again considered and another value from the corresponding domain is resorted to. A solution can be reached if no more variables are to be instantiated. When the backward step has no more variable to backtrack to it means that there is no solution whatsoever to the CSP. There are several techniques improving both the forward and the backward step:

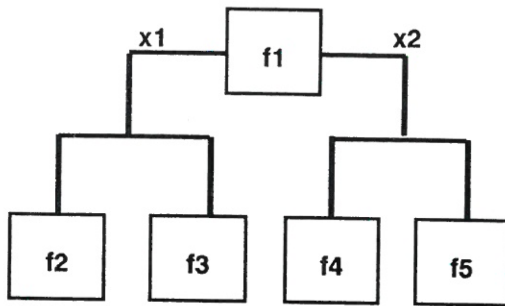
- computing and recording the way how the current instantiated variable set restricts the future assignments. This method (generally known as 'filtering') includes the Waltz algorithm described in [Waltz77];
- choosing the next variable to assign in a most profitable way (this applies only to methods that do not fix the variable instantiation order). A general rule is to search for the variable which, once instantiated, at maximum restricts the remaining search- space;
- choosing the value to assign (for the current variable). A fairly good strategy is to choose it, so that the number of options for future assignments should be maximized.

The backtrack may be optimized by:

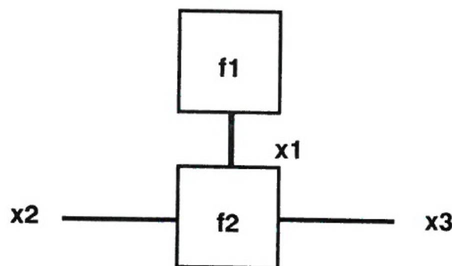
- using non-chronological backtracking (also known as 'intelligent backtracking' or 'dependency-directed backtracking') by jumping back to the variable that caused the failure (the culprit). The culprit may be computed either dynamically (when a backtrack occurs) or statically-before starting the solving process. The two methods present both advantages and tradeoffs. While the former guarantees the optimal 'back-jump', it is rather slow due to the extra calculation needed at each backward step; the latter is faster (the jump-to variable is pre-computed for every variable) but the back-jump may be 'shorter' than the optimal one;
- recording - as the solving process advances- the facts that caused failures, i.e. the subsets of variables-values pairs that induced constraint violations. TMS/ATMS mechanisms can be included here [De Kleer86a], as well as the learning techniques described in [Dechter90]. However, the (A)TMS's are usually too costly with respect to the time and space consumed, even for medium-sized CSP's;
- reducing, if possible, the constraint graph (hyper-graph for constraint arity greater than 2) to a tree structure that can be efficiently solved-solving time is linear in the number of variables (see [Freuder, Quinn85]).

3. A New Approach to CSP Solving

What the above enumerated methods have in common is that they all focus on **variables** rather than on **constraints**. In the 'classic' approach the variables are nodes and the constraints are arcs. Exploring the resulting hyper-graph - by variable instantiation - yields two negative aspects: 1. the set of applicable constraints has to be permanently recorded and maintained 2. this set evolves in a hard-to-control manner, often by leaps. For example, two successive instantiations may bring no additional constraint while a third variable assignment may enrich the applicable constraint set by 2,3 or more constraints. So, the granularity implied by this representation of the CSP problem is not high enough. Instead, if we consider the dual graph - constraints as nodes and variables as arcs, always representable as a graph - this problem never arises. For every constraint (node) we define the constraint-space as the k-dimensional discrete volume given by d^k (d = the variable domains cardinality, k = the constraint's arity). The elements in the constraint-space are Boolean values, TRUE if the constraint is satisfied for the assignation corresponding to that point and FALSE if not. These points are called T points (this stands for TruePoints) and Fpoints (this stands for False Points). Two important aspects appear: 1. choosing a Tpoint means simultaneous instantiation of several variables, more precisely, of the entire variable subset over which the constraint is defined; 2. during the graph exploration the next node to be considered is chosen at ease, independently of how constraints are interconnected via variables. Let us have a simple example:



The f_1 constraint defined over x_1,x_2; f_2,f_3 on x_1 and f_4,f_5 on x_2. After visiting the f_1 node we may explore the descending nodes in any order; we may choose for example the sequence f_2,f_4,f_3,f_5. This particular order corresponds to the following variable instantiation order: x_1,x_2,x_1,x_2 which makes no sense in the variable-focused approach. To capture the whole image of our CSP solving method let us consider the node to descendant switch in some other small example:



Supposing that the f_1 node was successfully reached, that is a Tpoint was found in f1's space. An implicit instantiation took for sure place for x1. We say that x_1 was CHOSEN in f_1. When moving down to f_2, with x_1 already assigned, we say that x_1 is IMPOSED on f_2. The search for a Tpoint in f_2 is going on in the (x_2,x_3) plan (or SUB-SPACE) yield by performing a cut into the f_2's volume (x_1,x_2,x_3). The cut corresponds to the value assigned to x_1 in f_1. The following algorithm for solving CSP results:

Procedure Solve_CSP (ConstraintList)

/* Solution is found by the side effect of Tpoint decoding */

L = ConstraintList

For all x in L **do**

fill space of x

F = first element in L

While (F is not null)

If (exists Tpoint in F's space) **then**

F = next element in L

else

F = Backtrack (F,L)

end

The Backtrack function of arguments current-node (constraint) and constraint-list returns the jump-back-to node if a failure occurs. The algorithm terminates in two cases: either no constraint is there to backtrack to, i.e. no (more) solution, or L is exhausted - a (new) solution has been generated (a Tpoint was found in each constraint's space). On evaluating the performance of this CSP solving method two phases are to be considered: 1. the constraint-space filling step 2. the search for a solution by exploring the constraint graph. For phase one, an elementary operation would be verifying a constraint against a certain assignation (an assignation of that variable over which the constraint is defined). The first step has a time and space complexity of $n*d^k$, where n is the total number of constraints in the CSP. This is less time consuming than if solving CSP by variable-oriented methods, which may be estimated to d_n; usually $k < n$, many CSP's are actually specified using only binary constraints, k=2. With regard to the occupied space, taking into account that the constraint space is made of Boolean values, a bit per stored value will suffice. To get a clearer picture, for a binary constraint defined over variables with domain cardinality = 128 the constraint space size will be of 128*128/8=2 Kbytes. Let us compare these results with the approach presented in [Guesgen91] - also a 'non-standard' CSP solving method. The quoted paper presents an algorithm for solving CSP that implies keeping a data structure for every Tpoint/Fpoint. Assuming only 100 bytes/structure (underestimate), for the example cited above there will be a memory need of 2 Mbytes for only one constraint space. In our opinion the method (although very sophisticated) is still unsuitable for today computing machines. Now, speaking about

the constraint space filling phase, another worthwhile aspect is to be mentioned: while this operation is under way, the number of Tpoints may be recorded as well. This makes it possible to define the constraint's STRENGTH as the ratio: Tpoints_no/TotalPoints_no, where Total Points_no = Tpoints_no + Fpoints_no = d^k.
Similar concepts are met with in [Fox83] where a sort of conditional probabilities are defined for both constraints and variables. In our case the constraint's strength is used in estimating a node's cost in the search-space ordering phase. The node cost is made by the Tpoints number in the current searched constraint plan (sub- space). The relation of the node cost is: node_strength* product_of_chosen_variables_cardinality, where a uniform distribution of Tpoints in the constraint's space was assumed. The above formula is analogous to the one used in finding the mass of a known volume given the material's density. Remember that the chosen variables are those that are not imposed by previous Tpoint-choosing, so that the node-cost should reflect and be influenced by both the way the already reached constraints are interconnected and the way the newly considered constraint (node) is connected to the reached nodes. The computed cost is therefore an almost exact measure of the dynamic complexity of the node. The 'almost' is introduced by the uniform distribution hypothesis; anyhow this uncertainty can be reduced by using techniques to be further described . In case of equal-cost conflict, a second criterion is used: the constraint with the highest connection degree is preferred. Another possible criterion will be the distance to the already reached nodes. The chosen node is that maximizing this distance. This causes a longer back-jump if a failure occurs during the search for a solution. With regard to the way the search-space is organized, a search tree is constructed in depth-first manner, using the above optimal criteria. Here are below other two features of our CSP solving module, thereby an easier problem declaration by the user and also an increase in solving efficiency are possible. 1. Elimination of included constraints. By 'included' constraints we understand those constraints applied to a subset X' of variables s.t. X' in X" where X" is the subset corresponding to another constraint. As an example, the constraint $f_1(x_1,x_2)$ is included in $f_2(x_1,x_2,x_3)$. Our system detects such situations and removes the included constraints in the space-filling phase. The gain is two-fold: 1. The operation is equivalent to a higher-order local consistency and causes a speed-up in finding the solution (the number of available nodes and Tpoints is reduced). On the other hand, a space saving of $m*d^k$ is feasible, where m is the number of constraints removed by inclusion 2. Automatic generation of the non-equal constraints. It takes place when two variables x_i and x_j share the same domain ('same' means here identity, not isomorphism). In such a case, whenever a value of the domain is assigned to x_i that value is no longer available to being assigned to x_j. This is equivalent to a binary non-equal constraint between x_i and x_j. According to the value of the :TestSameDomains option the system performs auto-detection and generates constraints of the mentioned type. Besides significantly simplifying the problem declaration, the non-equal constraint space being a diagonal matrix, the space may be filled in linear time - and not in $O(d^2)$.

4. Test Problems

Before describing the tests the CSP solving module goes through, let us point out some evaluation criteria. We have to consider the two phases of solving in our approach: the constraint-space filling and the search for a solution. The space-filling phase consists of performing $n*d^k$ consistency-checks. Consistency-check means the evaluation of a particular constraint under a certain assignation of its variable subset. In the second phase (the search space exploration) the elementary operation is the Tpoint-decoding- only Tpoints are considered during the search. Decoding a point from a constraint's space means finding out what is the variable assignation which corresponds to that point. Our product was tested against the following problems: the 'Zebra' problem, a crypto-arithmetic example, a 9 card-puzzle and a scheduler.

1. The 'Zebra' problem. It is too well-known to be reproduced here once more. However, this example may be found in the Appendix - we hope

our CSP-declaration syntax is enough clear for understanding the problem. In CSP terms the problem may be translated as follows: five groups of five variables each (nationalities, colours, drinks, cigarettes and pets) - a total of 25 variables. There are only five distinct domains and five elements (the houses) in each. Each domain is in correspondence with one of the variable clusters. The problem's constraints may be classified as 14 explicit constraints (stated in the problem) and 50 non-equal constraints generated by the system. The latter constraints are motivated by the variable domain-sharing. A number of 3 included constraints are eliminated in the space filling phase. The first phase may be ignored for this example because of the small domain cardinality. Theoretically, 1,600 consistency-checks should be performed. Only 350 are eventually performed because non-equal constraints are filled in linear time, without a real call on the constraint's function. The unique solution could be found after 279 decodings. The entire search-space was exhausted after 1,781 decodings. For comparison, data are available in [Dechter90]. If solved by means of a dependency-directed backtrack algorithm (the variables-as-nodes approach) the best variable ordering produces 1,234 consistency-checks till finding the solution (but **before** exhausting the entire search space). In the other 4 variable orderings this number varies from 20,000 to 90,000. The solving time was about 2 seconds (Golden Common Lisp 2.1 on IBM/PC 386 running at 25Mhz). We must say that the Lisp implementation used (an 1986 version) lacks some features of the Common Lisp standard. No bit-vectors, intensively used by the CSP solving system, are here. Bit-vectors were implemented **in** Lisp, making the solving time even more inefficient.

2. The SEND+MORE=MONEY problem. This rather simple crypto- arithmetic problem implies 8 digit-variables with 10 elements in their domains and 4 carry variables with 2 elements (0 and 1) in the domain. There are 4 explicitly stated constraints of arity 5 and 28 generated non-equal binary constraints. By eliminating the inclusions, a number of 11 constraints are removed yielding a total of 21 remaining constraints. About 12,000 consistency-checks are performed in the space-filling phase. The solution is reached after 141 decodings, the whole search space explorations end after 219 decodings. The problem was solved in less than 5 seconds.

3. The 9-card puzzle. Nine cards are given, the edges of these cards having one of four distinct colours. Each edge has an associated sign: the head or the tail of an arrow. The requirement is to find an arrangement for the cards (in a 3x3 matrix) so that all adjacent edges have the same colour and all arrows are correctly drawn. In terms of CSP, the problem may be stated as follows: 9 variables (positions where cards may be placed) with 9 elements (the cards set) in the domain. There is a single distinct - shared - domain. Due to the rotations with a 90° step, the cardinality of the domain increases up to 36. There are 48 constraints: 6 horizontal + 6 vertical + 36 non-equal constraints applied to each position pair. A number of twelve included constraints is removed. The constraint space filling consists of 47,000 consistency-checks. The search space is exhausted after 35,000 decodings (the first solution is reached after 18,000 decodings. The solving time was about 23 seconds.

4. A scheduler. The scheduling problem is defined via an interface function (**define-schedule**) allowing specification of activities, number of time samples and constraints. An extensible predefined constraint set was defined; for instance, the following relations between activities are grasped: Ai before|after Aj, Ai starts|ends-during Aj, Ai duration-is <duration>, Ai before|after-start|end-of Aj, Ai overlap|not-overlap Aj a.s.o. Tests involving a different number of activities and time samples were performed proving satisfactory execution times. The application and its results will be discussed in a future paper.

The four mentioned problems cover (with respect to the constraint net structure) all CSP problem types: - the scheduling problems have irregular, weak connected nets; - the zebra problem has a medium connected and quite irregular one; the other two problem types imply strongly connected, highly uniform nets. The system's response was satisfying in all the cases.

5. Integrating the CSP Solving Module into COPE

A first version of this module implemented the internal problem constraints as a new type of constraints in the general constraint representing and processing system COPE (see the previous paper on this issue). Experience invalidated this approach: major problems arose at value-propagation through the net, in that the CSP sub-net should have been first identified and only afterwards solved. A better solution was to encapsulate the whole CSP into a new COPE type of constraint, considering the variables of the CSP as the new constraint's cells. This approach is also conceptually sound, given that a CSP may be assimilated to a constraint with a case- described propagation that computes the rest of the cells according to the value propagated into one of them. For a CSP connected to other constraint types into a 'mixed' net the propagation algorithm is as follows: The propagated value is examined; if it is the special 'undefined' value, the CSP is solved considering the initial domains of the variables (cells). This is in fact the general CSP solving case. In other situations, a test is performed in order to verify if the propagated value is in the domain of the specified variable (cell); if not, an error is anticipated. After this test, the CSP is solved with the propagated cells' domain forced to a single value - the propagated one. After CSP solving, the values of the calculated cells are propagated into the net. A flexible mechanism will be implemented so that the propagation order for the computed cells should be easily controlled; dealing with multiple CSP solutions is another aspect to be taken into account. Note that for mixed nets containing CSP's the propagation has to be careful customized: according to the values propagated into the CSP meta- constraint and according to the order of computed cells further propagation, the CSP and the global net may have a solution or not. For not abandoning the ideas of the COPE system the Declare-CSP function (the analog of Declare-Constraint for the other COPE constraint types) creates and returns a prototype constraint. So, generating several instances of a certain problem and solving them under various environments become possible. The: Eval option is useful in such cases, making it possible the dynamic modification of the context within which the CSP is solved.

6.Conclusions

A new approach to CSP solving, based on pre-computation of the constraint spaces was proposed. The method is focused on the problem's constraints rather than on its variables. Promising results of some usual test-problems can be reported. Further exploration is necessary for identifying new criteria in structuring the search-space; combining the presented method with others (consistency algorithms, for example) in order to obtain larger flexibility and higher performance. At least two objections are likely to be raised against our approach 1. It is a 'brute-force' method. This is true, but it proved to be efficient enough, at least for medium-sized CSPs. 2. It is very space consuming. Also true, but taking into account the memory available on nowadays computers (8 or 16 Mbytes being a quite common capacity), we estimate that fairly large CSPs can be solved using the presented technique. For example, a binary constraint over variables with 1,000 elements in their domain would require less than 128 Kbytes to keep its space. Considering a memory of 16 Mbytes, it is possible to solve a CSP containing more than 120 constraints of this size.

Appendix: The Zebra problem.

```
;;; Domain initialization:

(defconstant HouseList

  '(House-1 House-2 House-3 House-4 House-5))

(mapc #'(lambda (house no) (setf (get house
'house-no) no))

      HouseList

      '(1 2 3 4 5))

;;; Create domains shared by the five clusters of
variables:

(defconstant Domain1 HouseList)

(defconstant Domain2 (copy-list HouseList))

(defconstant Domain3 (copy-list HouseList))

(defconstant Domain4 (copy-list HouseList))
```

```
(defconstant Domain5 (copy-list HouseList))
;;; Problem declaration:
(declare-CSP!
 Zebra
 :Vars&Domains
          (((Red Blue Yellow Green Ivory)  :On
Domain1
           :Eval T)
            ((Norwegian Ukrainian Englishman
Spaniard Japanese)  :On Domain2
           :Eval T)
       ((Coffee Tea Water Milk Orange) :On Domain3
           :Eval T)
       ((Zebra Dog Horse Fox Snails) :On Domain4
           :Eval T)
            ((Old-Gold Parliament Kools Lucky
Chesterfield)  :On Domain5
           :Eval T))
 :Constraints
        ((Zconstr1 :on Englishman Red)
          (Zconstr2 :on Spaniard Dog)
          (Zconstr3 :on Coffee Green)
          (Zconstr4 :on Ukrainian Tea)
          (Zconstr5 :on Ivory Green)
          (Zconstr6 :on Old-Gold Snails)
          (Zconstr7 :on Kools Yellow)
          (Zconstr8 :on Milk)
          (Zconstr9 :on Norwegian)
          (Zconstr10 :on Chesterfield Fox)
          (Zconstr11 :on Kools Horse)
          (Zconstr12 :on Lucky Orange)
          (Zconstr13 :on Japanese Parliament)
          (Zconstr14 :on Norwegian Blue))
 :Solutions :All
 :Trace T
 :TestSameDomains T)

;;; Some mnemonics:

(defmacro SameHouse (var1 var2)
 '(eq ,var1 ,var2))
(defun NextHouse (h1 h2)
 (= (get h2 'house-no)
   (1+ (get h1 'house-no))))
(defun NearHouses (h1 h2)
 (let ((o1 (get h1 'house-no))
      (o2 (get h2 'house-no)))
   (or (= o1 (1+ o2))
      (= o2 (1+ o1)))))
;;; Constraints:
(defun Zconstr1 (Engl Red) (SameHouse Engl
Red))
(defun Zconstr2 (Spaniard Dog) (SameHouse
Spaniard Dog))
(defun Zconstr3 (Coffee Green) (SameHouse
Coffee Green))
(defun Zconstr4 (Ukrain Tea) (SameHouse
Ukrain Tea))
(defun Zconstr5 (Ivory Green) (NextHouse Ivory
Green))
(defun Zconstr6 (OldGold Snails) (SameHouse
OldGold Snails))
(defun Zconstr7 (Kools Yellow) (SameHouse
Kools Yellow))
(defun Zconstr8 (Milk) (Eq Milk 'House-3))
(defun Zconstr9 (Norwegian) (Eq Norwegian
'House-1))
(defun Zconstr10 (Chesterfld Fox) (NearHouses
Chesterfld Fox))
(defun Zconstr11 (Kools Horse) (NearHouses
Kools Horse))
(defun Zconstr12 (Lucky Orange) (SameHouse
Lucky Orange))
(defun Zconstr13 (Jap Parliament) (SameHouse
Jap Parliament))
(defun Zconstr14 (Norwegian Blue) (NearHouses
Norwegian Blue))
```

REFERENCES

DECHTER, R., **Enhancements Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition**, ARTIFICIAL INTELLIGENCE, 41, 1989/90, pp. 273-312.

DE KLEER, J., **An Assumption-Based TMS**, ARTIFICIAL INTELLIGENCE, 28,1986, pp. 127-162.

DE KLEER, J., **Problem Solving with the ATMS**, ARTIFICIAL INTELLIGENCE, 28,1986, pp. 197-224.

DECHTER, R. and PEARL, J., **The Anatomy of Easy Problems: A Constraint- Satisfaction Formulation**, Proceedings of IJCAI '85, 1985, pp. 1066-1072.

DOYLE, J., **A Truth Maintenance System**, ARTIFICIAL INTELLIGENCE, 12, 1979.

FOX, M., ALLEN, B. and STROHM, G., **Job-shop Scheduling: An Investigation in Constraint-Directed Reasoning**, Proceedings of the second Conference of the AAAI, Pittsburgh, PA.,1982.

FOX, M., **Constraint Directed Search: A Case Study Of Job-shop Scheduling**, Technical Report, CMU-RI-TR-83-22, Univ. Carnegie-Mellon, 1983.

FREUDER, E.C., **Backtrack-free and Backtrack-bounded Search**, in L.Kanal and V. Kumar (Eds.) Search in Artificial Intelligence, SPRINGER-VERLAG, 1988, pp. 343-369.

FREUDER, E.C. and QUINN, M.J., **Taking Advantage Of Stable Sets Of Variables In Constraint Satisfaction Problems**, Proceedings of IJCAI '85, 1985, pp. 1076-1078.

FOX, M., SADEH, N. and BAYKAN, C., **Constrained Heuristic Search**, Proceedings of IJCAI89, Detroit, Ohio,1989.

MOTT, D. H., CUNNINGHAM, J., KELLEHER, G. and GADSDEN, J. A., **Constraint-based Reasoning For Generating Naval Flying Programmes**, EXPERT SYSTEMS, Vol. 5, No. 3, August 1988,pp. 226- 246.

MACWORTH, A. K. and FREUDER, E. C., **The Complexity Of Some Polynomial Network Consistency Algorithms For Constraint Satisfaction Problems**, ARTIFICIAL INTELLIGENCE, 25,1985, pp. 65-74.

NADEL, B.A., **Tree Search And Arc Consistency in Constraint Satisfaction Algorithms**, in L.Kanal and V. Kumar (Eds.) Search in Artificial Intelligence, SPRINGER-VERLAG, 1988, pp. 287-342.

ROSIERS, W. and BMYNOOGHE, M., **Empirical Study of Some Constraint Satisfaction Algorithms**, in Ph. Jorrand and V. Sgurev (Eds.) Artificial Intelligence II: Methodology, Systems, Applications, NORTH HOLLAND, 1987, pp. 173-180.

STEFIK, M., **Planning with Constraints (Molgen: Part1)**, ARTIFICIAL INTELLIGENCE, 16.1981, pp. 111-140.

SUSSMAN, G. J. and STEELE, G. L., **CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions**, ARTIFICIAL INTELLIGENCE, 14,1980, pp. 1-39.

BERLINER, H. and GOETSCH, G., **A Study of Search Methods: the Effect of Constraint Satisfaction and Adventurousness**, Proceedings of IJCAI, 1985.

GUESGEN, H.W. and HERTZBERG, J., **Some Fundamental Properties of Local Constraint Propagation**, Arbeitspapiere der GMD, April 1988.

GUESGEN, H.W., **Connectionist Networks for Constraints Satisfaction**, Arbeitspapiere der GMD, January 1991.

GUESGEN, H.W. and HERTZBERG, J., **Local Propagation in Networks of Filtering Constraints**, Arbeitspapiere der GMD, January 1988.

MONTANARI, U. and ROSSI, F., **Constraint Relaxation May Be Perfect**, ARTIFICIAL INTELLIGENCE, 48,1991.

WALTZ, D.L., **Generating Semantic Descriptions from Drawings of Scenes with Shadows**, Technical Report, MIT, Cambridge, MA., 1972.

MACWORTH, A.K., **Consistency in Networks of Relations**, ARTIFICIAL INTELLIGENCE, 8,1977.