Constraint Satisfaction - a Survey

Zsófia Ruttkay

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

e-mail: zsofi@cwi.nl

Constraint satisfaction has been used as a term to cover a wide range of methods to solve problems stated in the form of a set of constraints. As the general constraint satisfaction problem (CSP) is NP-complete, initially the research focused on developing new and more efficient solution methods, resulting in an arsenal of algorithms. Recently, much attention has been paid on how to finetune the use of this arsenal, and to be able to judge which methods are promising for a given problem or problem-type. In the last few years different generalisations of the classical CSP have got much attention too, allowing to model a wider range of every-day problems.

In this survey we introduce the classical CSP and the basic solution techniques as well as the ongoing research on the applicability of these methods and on extensions of the classical framework. After giving some introductory examples we define the most essential technical notions in order to explain different solution methods. First, we discuss constraint propagation algorithms, which transform the initially given CSP step by step to an equivalent, but smaller problem. Then we will introduce a family of constructive search algorithms, followed by methods exploiting the structure of the problem. Finally, we discuss the local and stochastic methods, also applicable to solve non-standard problems. The discussion of solution methods will be closed by addressing the issue of choosing a good algorithm for a given problem.

Many practical applications have essential characteristics which do not "fit into" the classical formalism of CSP. The extension of the problem definition and appropriate solution methods will be dealt with in the final chapter.

1. Constraint satisfaction problems

A constraint satisfaction problem prescribes some requirements for a finite number of variables in the form of constraints. The set of possible values - the domain - for each variable is finite. A constraint tells which value tuples are allowed for a certain subset of all the variables. A constraint can be given either explicitly, by enumerating the tuples allowed, or implicitly, e.g. by an algebraic expression. The solution of a CSP is an instantiation of all the variables for which all the constraints are satisfied. A CSP is solvable if it has at least one solution, otherwise it is unsolvable or overconstrained.

Solving a CSP is usually understood as the task of providing a single solution for the problem. However, there are cases when one would like to get all the solutions. In the case of constraint optimisation problems, the best solution is to be found, namely the one with the optimal value of a given optimisation function. In some situations one is not interested in the solutions themselves, but in the number of solutions. Particularly, if the problem is solvable or not. It has been shown that all these questions are equally hard, NP-complete problems (Macworth 1977). In this review by solving a CSP we understand the task of producing a single solution.

Before providing the formal definition of the CSP and its solution methods, let's look at some examples, also to illustrate the variety of the potential application fields.

1.1. 8-queens, graph colouring and practical applications

Famous test-problem popular also in the CSP world is the 8-queens problem: place 8 queens on the chess board such that they do not attack each other. In order to formulate this problem as a CSP, the location of the queens should be given by variables, and the "do not attack each other" requirement should be expressed in terms of a number of constraints. A simple way to do this is to assign a variable to each queen. As the 8 queens must be placed in 8 different columns, we can identify each queen by its column, and represent its position by a variable which indicates the row of the queen in question. Let x stand for the row of the queen in the i-th column. The domain of each of the variables x ,..., x is {1,2,..., 8}. For any two different variables the following two constraints must hold, expressing that the queens should be in different rows and on different diagonals:

x ¹ x ,

|x - x | ¹ |i - j| .

In this formulation of the problem, we have to find a solution out of the total possible instantiations of the variables, which is 8 . This formulation, though seems natural, does contain a trick: a part of the requirements of the problem is reflected in the representation, not in the constraints. We could have used the most straightforward representation, namely identifying the squares of the chess board by the 1,2,..., 64 numbers, and having 8 variables for the 8 queens all with the domain {1,2,..., 64}. In this case, the "different columns" requirement should be expressed too by constraints, and all the three types of constraints become more intrinsic to formulate. The total number of possible arrangements becomes as large as 64^8 , containing a configuration of queens multiple times due to the identification of the 8 queens. So we have many reasons to prefer the first representation over the second one.

It is true in general that a problem can be formulated as a CSP in a number of ways. The resulting CSPs may differ significantly considering the number and complexity of the constraints and the number of the possible instantiations of the variables, and thus may require very different amount of time and memory to be dealt with. Hence when modelling a problem as a CSP, one has to pay attention to different possibilities, and try to commit to the one which will be the easiest to cope with. The in-depth analysis of the different solution methods and of the characteristics of the CSPs may provide a basis to make a good choice. Several cases have been reported when a notoriously difficult problem could be solved finally as a result of change of the representation.

Both representations of the 8-queens problem are pleasantly regular: the domain of all the variables is the same, all the constraints refer to 2 variables, and for each pair of variables the same type of constraints are prescribed. Hence the 8-queens problem is not appropriate as a test case for solution algorithms developed to solve general CSPs. In spite of this intuitive observation, earlier the 8-queens had been a favourite test problem: there had been a race to develop search algorithms which were able to solve the problem for bigger and bigger n. (It is possible to construct a solution analytically.) This practice was stopped by two discoveries. On the one hand, Sosic (Sosic 1991) came up with a polynomial-time search algorithm, which was heavily exploiting the above mentioned special characteristics of the problem. On the other hand, by analysing the search space of the n-queens problem, it was shown that the general belief that "the bigger the n the more difficult the problem is" does not hold - in fact, the truth is just the opposite (Morris 1992).

Another, equally popular test problem is graph colouring: colour the vertices of a given graph using k colours in such a way that connected vertices get different colours. It is obvious how to turn this problem into a CSP: there are as many variables as vertices, and the domain for each variable is {1,2,..., k}, where k is the allowed number of colours to be used. If there is an edge between the vertices represented by the variables x and x , then there is a constraint referring to these two variables, namely: x ¹ x . Though for the first sight graph colouring may seem to be just as a toy problem as the n-queens, there are basic differences between the two problems. First of all, graph colouring is known to be NP-complete, so one does not expect a polynomial-time search algorithm to be found. Secondly, it is easy to generate a great number of test graphs with certain parameters, which are more or less difficult to be coloured, so the family of graph colouring problems is appropriate to test algorithms thoroughly. (We will return to the issue what makes a graph difficult to colour.) Finally, many practical problems, like ones from the field of scheduling and planning, can be expressed as an appropriate graph colouring problem.

Many other, sometimes surprising, fields of application raise problems which can be formulated as a CSP. Picture labelling is one of the basic tasks in image processing to analyse pictures and to recognise drawings. The lines in the picture should be labelled in a consistent way, choosing from a set of labels of objects which may occur in the drawing. That is, the assigned set of labels should properly define the objects given in the picture. The very first application of this kind -which is also one of the first applications of constraint technology - was given by Waltz (Waltz 1975), who could interpret line drawings of polyhedra. For the sake of simplicity, we restrict ourselves to line drawings of bricks. A line drawing is modelled as a CSP with variables corresponding to line segments in the drawing. Each variable should get a label from the set {+, -, ® , ¬ }. For each vertex in the drawing (junction of 2 or 3 line segments) there is a constraint. The allowed label tuples for the different kinds of junctions are given in Figure 1 (based on Huffman 1971).

It can be seen easily that in Figure 2 the labelling of the lines fulfils the constraints, hence this is a proper labelling (interpretation) of the drawing. The drawing given in Figure 3. cannot be labelled properly, that is, the corresponding

CSP has no solution. This is in accordance with the fact that the drawing cannot be interpreted as any view of a brick.

Many mechanical engineering design task can be given in the form of requirements with regard to certain parameters of the object to be produced (Frayman 1987, Nadel 1991). As from the point of view of functionality, safety, manufacturability, marketing and maintenance, different sets of requirements are formulated, which often cannot be fulfilled all, the successful practical applications have to be prepared to handle conflicts, to combine partial solutions, to allow hierarchical constraints (Bowen 1992). In the field of electrical engineering, CSP has been used to test VLSI designs (Hooker 1994).

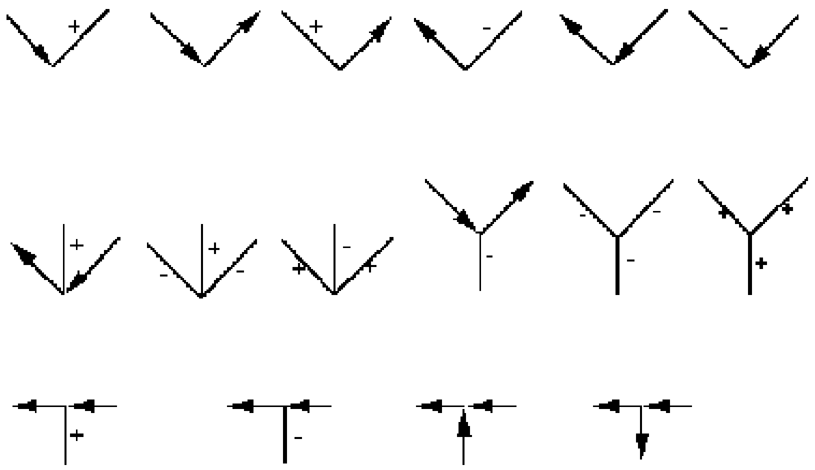


Figure 1. Proper labelling of line segments for different types of junctions.

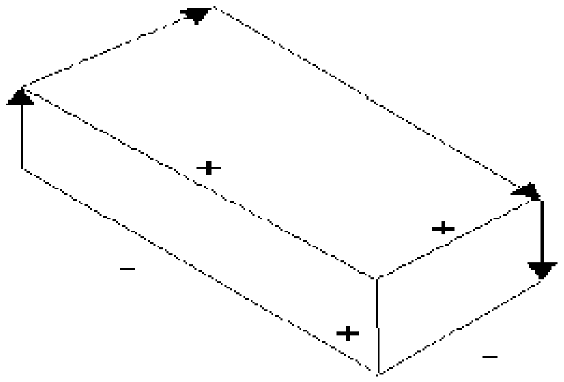


Figure 2. A consistent labelling of the line segments of a drawing.

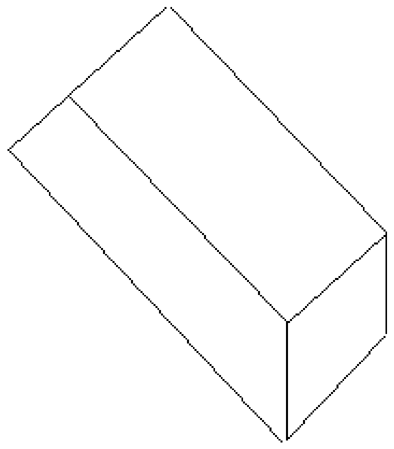


Figure 3. This drawing cannot be interpreted as a view of a brick - there is no solution for the labelling problem

Graphical user interfaces and interactive editors (Sutherland 1963, Borning 1995, Sannella 1994) assure certain layout by generating a solution of the CSP reflecting the layout criteria, each time some variables have been changed by the user or by an application program. The fields of application include transportation and scheduling (Fox 1989, Zweben 1994), natural language processing, robotics, molecular biology. For more applications, see a recent overview (Wallace 1996).

1.2. Definition and graph representations of the CSP

After the above intuitive description, in this chapter we give the formal definition for the CSP, and introduce certain characteristics to be referred to when discussing solution methods.

1. Definition

*A **constraint satsfation problem** (CSP) is a triple* $<X, D, C>$, where:

(i) $X = \{ x ,..., x \}$ is the set of **variables**.

(ii) $D = \{ D ,..., D \}$ is the set of **domains**. Each domain is a finite set containing the possible values for the corresponding variable.

(iii) $C = \{C ,..., C \}$ is the set of **constraints**. A constraint C is a relation defined on a subset {x ,..., x } of all the variables, that is

D x...x D Ê C . *

Given a (partial or complete) instantiation of the variables, the constraint C is **satisfied** if all the x ,..., x variables got a value and such that the corresponding value tuple belongs to C . A **solution** of a CSP is such a complete instantiation of the variables that all the constraints are satisfied. If for a CSP there is at least one solution, then the problem is **solvable**, otherwise **unsolvable,** or inconsistent, or overconstrained. The set of all possible complete instantiations, that is, D x...x D is often called the **solution space**, in the sense that the solution should be searched for in this space.

We emphasise that though a constraint is defined as a relation, it is not necessarily given explicitly, by enumerating the elements in the relation. It can be given implicitly, by any description which is sufficient to decide if the constraint holds for a value tuple or not. In case of variables with numerical domains, a constraint can be given as an equation, inequality, or the prescription that the value of two variables values should be relative prime numbers. A constraint can be given by a function or logical predicate, as well.

A constraint is **loosened** or **relaxed** if there are further element(s) added to the relation. If elements are removed from the relation, then the constraint is **tightened**. The variables of a constraint C are the variables in the set v(C)={x ,..., x }. It is also said that constraint refers to these variables, or it constraints these variables. (Note that in our definition of a CSP for sake of simplicity we did not include the variables of a constraint explicitly. In more strict and formal discussions they often do.) The **tightness of a constraint** is given by the proportion of the number of elements in the constraint and the number of all possible instantiations of the referred variables. The **cardinality of a variable** is the number of constraints referring to the variable.

If v(C) has one element, then C is **unary constraint**; if v(C) has two elements, then the constraint is **binary constraint**. A CSP is a **binary CSP**, if all its constraints are unary or binary.

Binary CSPs play a special role, as any CSP can be transformed into an equivalent binary CSP. Namely, if in a given CSP, C is a k-ary (k>2) constraint, then we may replace it by an y variable and the B ,..., B binary constraints. The domain of the y variable is the set of those k-tuples for which C holds. B refers to x and y, and B holds if the value assigned to x is the same as the value in the i-th position of the instantiation of y. By eliminating all the non-binary and non-unary constraints in such a way, the resulting problem is a binary CSP. It can be seen easily that this problem is solvable if and only if the original one is solvable, and there is a one to one correspondence between the solutions of the original CSP and the derived binary one. No question, a binary problem has a nicer and simpler structure than a general one. As we will see later, there are solution methods which exploit this specific structure and are only applicable to binary problems. However, as any problem can be transformed to an equivalent binary one, after all these solution methods can be considered as general. In practice, nevertheless, one should check carefully if it is worth to deal with the binary equivalent of a given general CSP, as this usually implies a problem with many additional variables with large domains, hence a problem with a considerably bigger solution space than that of the original problem.

The structure of a binary CSP can be visualised in a straightforward way by a graph, the vertices of which correspond to the variables, and two vertices are connected iff there is at least one constraint referring to the corresponding variables. This graph is called the **constraint graph** of the binary CSP. The constraint graph of the 8-queens problem is the complete graph with 8 vertices. The constraint graph of a graph colouring problem is isomorphic with the graph to be coloured.

The concept of constraint graph can be defined for general CSPs as well. Constraints are not represented by edges, but by the set of vertices corresponding to variables of the constraint. Such a hyper-edge can be shown by drawing a closed curve around the vertices involved. For a general CSP the constraint graph is thus a hypergraph.

Another presentation of a CSP is its **primal graph**. It is a non-directed graph, with vertices corresponding to the variables of the problem. There is an edge between two vertices iff there is at least one constraint in the problem such that its referred variables contain the two ones corresponding to the vertices. For a binary CSP, the constraint graph and the primal graph are the same.

The **dual graph** of a CSP is a non-directed, labelled graph with vertices corresponding to the constraints of the problem. Two vertices are connected iff the two constraints represented by the vertices share at least one variable. The edge is labelled by the set of variables shared by the two constraints. In Figure 4 the different graph representations are shown for the picture labelling problem given in Figure 2.

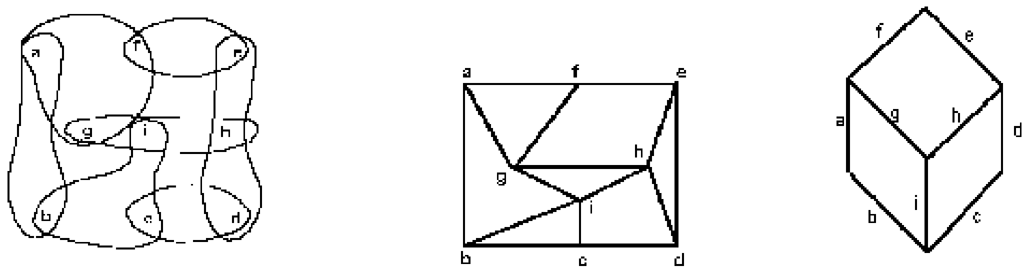
The structure of a CSP is characterised in terms of topological features of its primal graph. A CSP is **connected**, if it primal graph is connected.

The **width** of a CSP is the width of its primal graph, which is defined as follows. Let's consider an ordering of the variables, x , ...x . The **width of a variable** x **at the given ordering of all the variables is the number of edges in the primal graph which connect** x with another variable preceding it in the ordering. The width of an ordering is the maximum of the variables at the given ordering. The **width of a primal graph** is the minimum of the width of all possible orderings of the variables.

As we will see, CSPs with a tree primal graph are easy to solve. It is easy to prove that a graph which is a tree has width 1.

If all the variables have got a value, then we talk about complete instantiation of the variables. Otherwise the variables are partially instantiated. If we assign a value to an uninstantiated variable in a partial instantiation, then we **extend the partial instantiation**. We will refer to an instantiation of a variable as <x,v>. The **projection of a constraint** C on the variables {x ,..., x } , denoted by

P(C , {x ,..., x }), is the set of all (v ,..., v) value h -tuples for which it holds that the (<x ,v >,..., <x ,v >) partial instantiation can be extended in such a way that the constraint holds. The projection of several constraints on a set of variables is the intersection of the projection of the individual constraints. Particularly, the projection of all the constraints on a set of variables - denoted by P(*C*, {x ,..., x }) - is also called as the projection of the solution set on the variables. A **partial instantiation is consistent** if all the constraints which refer to instantiated variables only, are satisfied. An <x,v> instantiation of a variable is compatible with a partial instantiation if the extension of the partial instantiation with <x,v> is consistent.



a) b) c)

Figure 4. Graph representations of the consistent labelling problem given in Figure 2. a) the constraint graph b) the primal graph c) the dual graph

A CSP is usually formulated in such a way that the domain D contains other elements too than the ones in P(*C*, *{x }*). Removing those elements from D has no effect on the solution set. However, it decreases the size of the solution space, which is advantageous when searching for a solution. The same can be told of constraints as well. Usually, a constraint contains more tuples than the projection of the solution set on the variables of the constraint, and those additional value tuples can be eliminated without having an effect on the set of solutions. By tightening a domain and/or a constraint in such a way the CSP is **reduced** to an equivalent one. A CSP is **minimal** if it cannot be reduced. A minimal CSP can be considered as the ideal representation of the solution set, as there are no redundant values left in the domains and in the constraints. The task of producing the minimal equivalent of a CSP is also NP-hard (Freuder 1978), thus in general it is not a feasible scenario to reduce the problem to its minimum and then solve it. However, as we shall see, a certain amount of reduction can pay off.

1.3. Solution methods

The methods to generate a solution for a CSP fall into three classes. The first class includes the variants of backtracking search. These algorithms construct a solution by extending a partial instantiation step by step, relying on different heuristics and using more or less intelligent backtracking strategies to recover from dead ends.

As we saw in the previous chapter, the reduction of a problem is advantageous, resulting in a smaller solution space to be searched. The second class includes the so-called constraint propagation algorithms, which do eliminate some non-solution elements from the search space. In general, these algorithms do not eliminate all the non-solution elements, hence, they do not produce a solution on their own. They are used either to pre-process the problem before another type of algorithm is applied, or interwoven with steps of another kind of - e.g. backtracking search - algorithm to boost its performance.

Finally, the structure-driven algorithms exploit the structure of the primal or dual graph of the problem. There are very different algorithms in this class, including ones which decompose the initial CSP into loosely-coupled subproblems, which can be solved by methods from the previous two classes. Hence, structure-based methods can be also coupled with some other types of algorithms.

All the algorithms from the above three classes investigate the solution space systematically. Hence all those algorithms of the above classes which are meant to find a solution, in theory really do the job as long as there is a solution. These algorithms are:

(i) **sound**, that is if they terminate with a complete instantiation of the variables then it is a solution;

(ii) **complete**, that is capable to investigate the entire search space and hence find all the solutions.

These requirements seem to be very essential, however, often one has to be satisfied with algorithms which do not fulfil one or both of them. The systematic search algorithms require exponential time for the most difficult CSPs. A CSP is difficult if (almost) the entire search space has to be investigated before finding a solution or concluding that the problem has none. If the search space is large, then it may take days or weeks to run a complete and sound algorithm. This can be forbidding in case of applications where a solution can be used only if provided within a short time. In such cases a compromise is made, by using an algorithm which provides an answer fast, but the answer is not guaranteed to be a solution. However, it is "good enough" in the sense that not all the constraints are satisfied, but the number of non-satisfied constraints and the degree of violations can be accepted. Though such algorithms cannot be used to generate all the (good) solutions for sure, usually it is possible to generate several quite different "almost solutions" (if they exist). The so-called local stochastic search algorithms have in common that they explore the solution space in a non-systematic way, stepping from one complete instantiation to another, based on random choices, and may navigate on the basis of heuristics, often adopted from systematic search methods. In the recent years such algorithms have been used with success to solve large practical problems, and they are suitable to handle CSPs extended with some objective function to be optimised. We will devote a chapter to three algorithms of this kind: local search, genetic algorithms and neural networks.

One may try to tackle the difficulty of solving CSPs by another approach than trying to forge new and better algorithms. The exponential time complexity of a CSP solving algorithm refers to the worst case, i.e. the most difficult CSP instances. The same algorithm could be applied with success for non-difficult instances. Should one be able to judge the difficulty of CSPs in advance, based on some characteristics of the problems, one could try to avoid difficult instances, and could get a forecast about the resources needed to solve a problem at hand. How to make such a forecast? As a closing topic of the discussion of solution methods, we will quote a couple of interesting results form the intense research having been carried out in the past years.

2. Constraint propagation

By eliminating redundant values from the problem definition, the size of the solution space decreases. Reduction of the problem can be done once, as pre-processing step for another algorithm, or step by step, interwoven with the exploration of the solution space by a search algorithm. In the latter case, subsets of the solution space are cut off, saving the search algorithm the effort of systematically investigating the eliminated elements, which otherwise would happen, even repeatedly. If as a result of reduction any domain becomes empty, then it is known immediately that the problem has no solution.

One should be careful with not spending more effort on reduction than what will "pay off" in the boosted performance of the search algorithm to be used to find a solution of the reduced problem. The reduction algorithms eliminate values by propagating constraints. The amount of constraint propagation is characterised by the consistency level of the problem, hence these algorithms are also called consistency-algorithms. The iterative process of achieving a level of consistency is sometimes referred to as the relaxation process, which should not be mixed up with relaxation of constraints.

Constraint propagation has a long tradition in CSP research. Below we introduce the most well-known and widely used algorithms.

2.1. Node- and arc-consistency

A CSP is **node-consistent,** if all the unary constraints hold for all the elements of the domains. The straightforward node-consistency algorithm (NC), which removes the redundant elements by checking the domains one after the other has O(dn) time complexity, where d is the maximum of the size of the domains.

A CSP is **arc-consistent**, if for any u value from the domain of any variable x any binary constraints which refers to x can be satisfied. The Domain_Reduction procedure given below eliminates those elements from the domain of a variable x for which a given binary constraint cannot be satisfied. (The given pseudo-programs are based on ones in (Tsang 1993)).

```
Procedure Domain_Reduction (<x,y>, domains, constraints)

1. reduced ¬ FALSE

2. domain_x ¬ get_domain(x, domains)

3. domain_y ¬ get_domain(y, domains)

4. while not all_checked(domain_x) loop

5. v ¬ pick_not_checked_value(domain_x)

6. if not good_value(v, domain_y, constraints) then

7. domain_x ¬ domain_x \ {v}

8. reduced ¬ TRUE

9. endif

10. endloop

11. return (reduced)

end
```

Running the Domain_Reduction procedure for all the binary constraints, it is not sure that the reduced problem is arc-consistent. Let’s assume that the edge <x,y> is on turn, after the edge <z,x> was processed. When reducing the domain of the variable x we may eliminate a value which assured that the binary constraints referring to z and x could be satisfied for a value not eliminated earlier form the domain of z. Thus the edge <z,x> should be checked again, and the Domain_Reduction should be reinvoked until it does not reduce further any of the domains. The very first AC-1 algorithm has been followed by the algorithms AC-2,..., AC-7, each improving the bookkeeping of the changed domains and the administration of the iterations. Below we give the widely-used AC-3 algorithm:

```
Procedure AC-3 (variables, domains, constraints)

1. NC (variables, domains, constraints)

2. arcs ¬ binary_constraints(constraints)

3 while arcs ¹ Æ loop

4. <x,y> ¬ pick_one_element(arcs)

5. if Domain_Reduction(<x,y>, domains, constraints) then
```

```
        6. arcs ¬ arcs È (binary_constraints(constraints) Ç {<z,x> | z¹ y})
    7. endif
    8. endloop
    9. return (domains)
    end
```

The time complexity of AC-3 is O(d e), memory complexity O(e+nd), where e is the number of binary constraints. AC-4 improves AC-3 by checking if in line 6 the just eliminated value did destroy the arc consistency of edge <z, x>. This modification results in an algorithm with O(d e) time- and space complexity. Though we shall address the issue of interpreting worst-case complexity estimates in Section 6, we must note now that testing AC-3 and AC-4 on a large body of CSPs, on average AC-4 did not run faster than AC-3 (Wallace 1993). Another improvement of AC-3 is the algorithm AC-5 (Van Hentenryck 1992), which exploits the semantics of special types of constraints. The AC-7 algorithm (Bressiere 1995) uses the symmetry of binary constraints.

Assuring node- and arc-consistency pays off for sure for binary problems with tree structure. This is stated by the following, easy to prove theorem:

2. Theorem (Freuder 1982)

*If a binary CSP has tree structure, and it is node- and arc-consistent, then a solution can be constructed without backtracking. **

For the success of a backtrack-free search it is sufficient to assure that there is a value for the variable to be instantiated in the current step which is consistent with the earlier produced partial instantiation. Based on this observation, the condition in the above theorem can be weakened: it is sufficient to require that instead of arc consistency a weaker, so-called directional arc-consistency holds for the problem.

For a CSP **directional arc-consistency** holds along a given ordering of the variables, if for any variable x, for any value u from its domain which satisfies the unary constraints on x, taking any variable y succeeding x in the ordering, there is a value v in the domain of y such that the binary constraint on x and y is satisfied for x=u, y=v.

Dechter and Pearl (1988) not only sharpened the theorem by Freuder, but since then have coined a series of other directional consistency concepts and have shown how the different directional consistencies improve the efficiency of the uninformed search algorithm.

2.2. k-consistency

Arc-consistency can be also understood as telling something about how far a partial solution can always be extended. Namely, any partial solution containing only one instantiated variable can be extended by instantiating any second variable to a properly chosen value. Applying the same principle for more variables, we arrive at the concept of k-consistency.

3. Definition

A CSP is **k-consistent**, if any consistent instantiation of any k-1 variables can be extended by instantiating any one of the remaining variables. *

Note that according to the definition, k-consistency does not imply r-consistency for any r such that r<k. This is assured by the definition of strong k-consistency.

4. Definition

A CSP is **strongly k-consistent**, if 1, 2, ..., k-1 and k-consistent. *

For binary CSPs there is yet another type of consistency used:

5. Definition

A binary CSP is **path-consistent**, if for any path in its constraint graph it holds that if the assignments of the starting and ending variables are consistent, then this can be extended to a consistent partial instantiation by assigning values to the remaining variables along the path. *

It is an easy exercise to prove the following theorem:

6. Theorem

A binary CSP is strongly 3-consistent if and only if it is path-consistent. *

Similarly to the family of arc-consistency algorithms, there is a family of algorithms to achieve path-consistency of a binary CSP. These algorithms also assure node- and arc-consistency. The best of them, PC-4 which is the analogue of AC-4, has O(d n) time- and space-complexity. A weaker, directed variant of path-consistency is also used (Dechter 1988).

It is important to understand clearly the significance of consistency. A CSP which is k-consistent, is not necessarily solvable, and in the other way around, the solvability of a problem does not imply any level of consistency, not alone 1-consistency. The consistency as a feature of a problem does guarantee that certain values and value h-tuples which are not in the projection of the solution set have been removed form the domains and constraints. The level of consistency, k indicates for what h-values has this been done. It is true, however, that if a problem is strongly k-consistent, then taking any k variables in any order, they can be instantiated without backtracking one after the other in such a way that the instantiation of the k variables is a partial solution, assuming that one exists. This partial instantiation is then a solution of the subproblem which is gained by restricting the problem to those variables.

7. Theorem

If a CSP is strongly k-consistent, and none of the domains is empty, then restricting the problem to any set of k *variables, this subproblem is solvable and the solution can be constructed with uninformed search, without backtracking. **

This theorem too has a directed version due to Dechter and Pearl, which assures that the solution of a subproblem can be constructed without backtracking assuming a given ordering of the instantiation of the variables. To achieve this, instead of a uniform level strong k-consistency it is enough to require **adaptive-consistency**, that is the right level of consistency for the subproblems one is to solve one after the other, which assures that the variable on turn can be instantiated without backtracking.

The practical value of the above theorems depends on how the effort needed to reduce the given problem to one with a certain level of consistency is related to the gain in resources needed to solve the reduced problem. To achieve k-consistency for a big k is expensive, as the time complexity of the k-consistency algorithm is beyond O(d^{1+a}). For adaptive consistency, the same worst-case estimate is O(d), where a denotes the so-called induced width of the given CSP - the computation of which is itself NP-hard. However, for a given CSP these algorithms may work much better than the worst-case estimates guarantees. The other factor, the gain is also unclear, as no good in-advance estimates exist. When facing the "how much reduction to be done" dilemma for a given CSP, there remains the study of empirical results (Frost 1995) and the general principle that it is profitable to propagate tight constraints, as loose ones do not cause much backtracking anyway.

3. Search algorithms

A CSP can be solved by systematically exploring the solution space by an uninformed search. The algorithms instantiate variables one after the other in such a way that the partial instantiation is always consistent. If this is not possible for a variable on turn, that is, all the possible values are in conflict with some earlier assignment, then backtracking takes place.

3.1. Chronological backtracking

The simplest search scenario is when the order of variables as well as the order of values to be considered next is fixed, and if a dead end occurs, then the latest instantiation is reconsidered. In this uninformed, depth-first search no heuristic is used for any of the choices made during exploration. The pseudo code of the chronological backtracking (CB) algorithm is given as a basis to discuss possible improvements:

```
Recursive procedure CB (free_variables, instantiation, domains, constraints)
1. if free_variables= Æ then return(instantiation) endif
2. x ¬ pick_one_variable(free_variables)
3. domain_x ¬ get_domain(x, domains)
4. while domain_x ¹ Æ loop
5. v ¬ pick_one_value(domain_x )
6. if consistent(instantiation È {<x,v>}, constraints) then
        7. solution ¬ CB(free_variables \ {x}, instantiation È {<x,v>}, domains, constraints)
8. if solution ¹ Æ then return(solution) endif
9. endif
10. endloop
11. return (Æ )
```

```
        end
```

Running the above procedure with free_variables ¬ *X*, instantiation ¬ Æ , domains ¬ *D*, and constraints ¬ *C* as initial values, the procedure will return a solution - the one found for the first time -, if the problem is solvable, Æ otherwise. The worst case is when the algorithm has to generate and check all the complete instantiations to find the only solution or to conclude that there is none. This requires $O(d^n m)$ steps and O(dn) memory.

The CB algorithm can be improved at different points in the following ways:

(a) Whenever a choice has to be made, some heuristic is to be used instead of the fixed lexicographical ordering given by the description of the problem.

(b) After each variable instantiation, a part of the search space not containing solutions is identified and discarded from further investigation.

(c) Analysing the structure of the problem, the chronological backtracking is replaced by some intelligent backtracking.

(d) The algorithm learns while exploring the search space: both negative and positive conclusions on partial instantiations are remembered and reused later if appropriate, instead of redoing the necessary checks again.

3.2. Ordering heuristics

Variable instantiations should be done in such an order that we try to avoid deep backtracking in course of the search. A deep backtracking means that a big part of the variable instantiations have to be undone, often repeatedly many times. (This annoying phenomena in search got a distinct name: thrashing.) The reason for this is that the variable, which cannot be instantiated properly for a big set of partial instantiations, is dealt with too late. Hence, intuitively, one would deal with the critical variables first. Different heuristics are used to judge how critical variables are, based on the number of possible values for the variable and the already satisfied and remaining constraints. The heuristic-based selection, whatever heuristic is used, replaces the uninformed pick_one_variable process. If the order of variables is set in advance, then we talk about static variable ordering. Otherwise, in the case of dynamic variable ordering, the next variable to be instantiated depends on the current state of the search.

The **fail first principle** (FFP), which is a general search heuristic, in our case of variable selection means that variables with the least possible values should be instantiated first. This principle can be applied in a static way, if the initial domain of the variable is taken into account, or in a dynamic way, if the domains of uninstantiated variables may get reduced due to the already committed instantiations.

The **minimum width ordering** (MWO) heuristic is based on the idea that if a variable does not depend on many earlier instantiated variables, then it will be easier to assign an appropriate value to it. Hence the smaller the width of an ordering is, the fewer backtracking can be expected. The heuristics takes a static ordering of the variables, namely one with minimal width.

If the width of a CSP is w, then taking an ordering of the variables with this minimal width, for each variable on turn at most w-1 earlier instantiated variables are to be taken into account. The strength of the heuristic depends on the consistency level of the problem, as illuminated by the following theorem.

8. Theorem (Freuder 1982)

If for a CSP with level of strong consistence s and width w holds that s>w, then a solution can be produced without backtracking, by instantiating the variables in the order corresponding to the width of the problem. *

The MWO heuristic thus does guarantee backtrack-free search, assuming that the problem has a sufficient level of strong-consistency. But even if the necessary condition does not hold, that is w³ s, then the magnitude of (w-s) can be used to judge the amount of backtracking to happen: the smaller the difference is, the less backtracking is expected.

To produce the minimal width ordering of the variables requires O(n) steps. O(n) steps are sufficient to order the variables by cardinality. The **maximum cardinality ordering** (MCO) heuristic, which takes the variables in the decreasing order of cardinality, can be considered an approximation of the MWO heuristic. This cheaper heuristic has been used with success.

Another heuristic is based on the idea that variables which constraint each other should be close to each other in the order of instantiations. This principle is reflected in the **minimum bandwidth ordering** heuristic (MBO), which takes a variable ordering with minimal bandwidth. The bandwidth of an ordering is the maximum of the bandwidth of the vertices in the ordering. The bandwidth of a vertex in an ordering is the maximum distance of the vertex from all the vertices in the ordering which are constrained by the given vertex.

We have seen that it is advantageous to consider the most critical variables to be instantiated first. When deciding about the order of assigning values to a variable on turn, the situation is just the opposite: one would like to take a value which will not have to be reconsidered later on, due to backtracking. Hence, the "most promising" values should be tried first. "Most promising" in the sense that the partial instantiation resulting by assigning the value to the variable on turn, the partial instantiation can be extended further to a solution. A common measure to judge the "promise" of a value is based on the number of values not in conflict with the value in question, for each uninstantiated variable. The value which leaves the most options for each uninstantiated variable is the best. This heuristic is known as **minimum conflict first** (MCF) heuristic.

Both for variable and value ordering there is the question of breaking ties, that is, selecting one from several equally good candidates. This is done either in a deterministic way, based on lexicographic ordering or applying another principle to differentiate further between the candidates; or by choosing randomly from the equally good candidates. It has been observed that random choice is better than uninformed deterministic choice.

3.3. Intelligent backtracking

We illustrate two weaknesses of uninformed search by an example. Let us assume that the CSP to be solved has the variables x, y and z, each with the domain {1,2,3}. The constraints to be satisfied are: y £ z, xz ³ 4. Let us assume that in the search process the x=1, y=1 instantiations have taken place. Now when trying to instantiate z, it turns out that for all the possible values for z the xz ³ 4 constraint is violated. Uninformed backtracking will reconsider the variable y in vain, ending up in the same dead-end situation, without noticing that the cause of the dead end is obviously in the value assigned to x, the very variable involved in the violated constraint. On the other hand, when after having backtracked to x, y is re-instantiated again to 1, and a value for z is to be chosen, the y £ z, constraint will be checked again, though earlier it was found that if y=1 then none of the variables from the domain of z were in conflict with the value of y, hence this constraint should not be checked again if y=1.

These shortcomings, due to chronological backtracking and blind constraint checking, can be avoided by:

(a) backtracking to such a variable which can be the cause of the detected dead-end situation;

(b) when extending a partial solution again after backtracking, some earlier proven partial instantiations which are not effected by recent changes are re-used instead of re-generating and testing the assignments again.

Depending on the policy used to pick the variable to backtrack to, and on the amount of learning taking place during the exploration of the solution space, different **intelligent backtracking** methods are defined.

In the discussion of possibilities for intelligent backtracking we will assume binary CSP, and that the variables are instantiated in a fixed order. We will refer to the place of occurrence of a variable in this ordering as its level. Given a feasible partial instantiation, the conflict set for an x=v assignment is the set of those variables in the partial instantiation which got such a value that at least one constraint is violated when x=v.

Below we give the code for the simplest realisation of intelligent backtracking, the so-called **backjumping** (BJ). In a dead-end situation, all the possible values for the variable x are in conflict with at least one earlier instantiation. The algorithm takes the union of the conflict sets belonging to the possible values, and backtracks to that variable x in the union which was instantiated the most recently. Hence the variables x , ... x are jumped over, instead of doing chronological backtracking. Backjumping makes sense only if the dead-end situation was discovered at x . If an x =v instantiation was successful, but later on it turns out that backtracking has to take place, chronological backtracking will be performed. This is assured by the Where_to_Backtrack procedure.

```
Recursive procedure BJ(free_vars, inst, domains, constraints, dl, cl),
1. if free_variables = Æ then return(inst) endif
2. x ¬ pick_one_variable(free_vars)
3. set_level(x, dl, cl)
4. domain_x ¬ get_domain(x, domains)
5. result ¬ Æ
6. while (domain_x ¹ Æ and not (backtrack_to(result, bl) and bl< cl) ) loop
7. v ¬ pick_one_value(domain_x )
8. if consistent(inst È {<x,v>}) then
                              9. result ¬ BJ(free_vars\{x}, inst È {<x,v>}, domains, constraints, dl, cl+1)
10. if not (backtrack_to(result, bl) then
11. return (result)
12. endif
13. endif
14. endloop
15. if (backtrack_to(result, bl) and bl< cl) then
16. return (bl)
17. else bl ¬ Where_to_Bactrack(x, inst, domains, constraints, cl)
18. return(bl)
19. endif
end
```

```
Procedure Where_to_Backtrack(x, inst, domains, constraints, dl, level)
1. jump_level¬ -1
2. domain_x¬ get_domain(x, domains)
2. for each v Î domain_x loop
3. temp ¬ level-1, no_conflict ¬ TRUE
```

```
4. for each <y, w> Î inst loop
5. if not consistent ({<y, w>, <x, v>}) then
6. no_conflict ¬ FALSE
7. temp ¬ min(temp, get_level(y, dl))
8. endif
9. endloop
10. if no_conflict then return (get_level(x, dl)-1)
11. else jump_level¬ max(jump_level, temp) endif
12. endloop
13. return(jump_level)
end
```

Conflict-directed backjumping (CBJ) is an improved variant of the above algorithm, which jumps back to a potential cause of a conflict whenever a variable instantiation has to be undone, not only when a dead-end is detected.

The procedure Where_to_Backtrack gains valuable information to decide about the level of backtracking, however, this information gets forgotten, not used any more. Though feasible({<y, w>, <x, v>}) remains true and could be re-used as long as <y, w> is part of the partial instantiation. **Backchecking** (BC) exploits this fact, by eliminating all values from the domain of the current variable to be instantiated which are not compatible with some earlier assignment.

Backtracking does not register which instantiation forbade a value. Hence if the level of backtracking exceeds the level of the variable, the complete domain is reconsidered and checked again. **Backmarking** (BM) improves backchecking at this point, by keeping administration of the instantiations in the partial solution which exclude a value form the domain of a variable x, and compares this to the depth of backtracking since the last instantiation of x.

The procedure Where_to_Backtrack does not give information about the set of earlier instantiations which together forbid all the possible values for x and hence force backtracking. E.g. if the conflict sets belonging to the different possible values are such that at least one of the y and z variables is member of each set, then these two variables with their corresponding instantiations cannot be part of a solution, as they do not leave a compatible value for x. If it also holds that neither y nor z occurs in all conflict sets, then {y, z} is a minimal conflict set. Thus this set of instantiations as a "nogood set" can be put on the list of forbidden instantiations, which cannot be extended to a complete solution. Naturally, there can be more than one nogood sets which all should be excluded from consideration. The learning nogood labels or recording nogood constraints algorithm works on this principle. That is, after having generated all the conflict sets for the values for the current variable, it produces the minimal covering sets of the conflict sets. These minimal covering sets are the nogood sets. The generation and administration of minimal covering sets is referred to as deep learning, while the opportunistic recognition and registration of (not necessarily minimal) covering sets is referred to a shallow learning.

3.4. Lookahead algorithms

Let us assume that when searching for a solution, the variable x is given a value which excludes all the possible values for the variable y. In case of uninformed search this will only turn out when y will be considered to be instantiated. Moreover, in case of chronological backtracking, thrashing will occur: the search tree will be expanded again and again till y, as long as the level of backtracking does not reach x. Both anomalies could be avoided by recognising that the chosen value for x cannot be part of a solution as there is no value for y which is compatible with it. Lookahead algorithms do this, by accepting a value for the variable on turn only if after having looked ahead, it could not be seen that the instantiation would lead to a dead-end. When checking this, problem reduction can also take place, by removing the values from the domain of the future variables which are not compatible with the current instantiation. The algorithms differ in how far and thoroughly they look ahead and how much reduction they perform.

Forward checking (FC) checks the satisfiability of the binary constraints, and removes the values which are not compatible the current variable’s instantiation.

```
Recursive procedure FC(free_vars, inst, domains)
1. if free_variables= Æ then return(instantiation) endif
2. x ¬ pick_one_variable(free_vars)
3. domain_x ¬ get_domain(x, domains)
4. while domain_x ¹ Æ loop
5. v ¬ pick_one_value(domain_x )
6. if consistent(inst È {<x,v>}) then
                    7. reduced_domains ¬ Reduce_Domains(<x,v>, free_vars\{x}, domains)
8. if not there_is_empty_domain(reduced_domains) then
                              9. solution¬ FC(free_vars\{x}, inst È {<x,v>},
                              reduced_domains)
10. if solution¹ Æ then return(solution) endif
11. endif
12. endif
13. endloop
14. return (Æ )
end
```

The Reduce_Domains procedure does basically the same as the Domain_Reduction introduced in 2.1, but returns the reduced domains.

Forward checking can be performed for general constraints too, assuring that only such partial solutions are generated for which all the constraints with one uninstantiateted variable can be satisfied one by one.

The **full lookahead** or arc-consistency lookahead extends this with assuring arc-consistency of the remaining problem to be solved. That is, the domain of the uninstantiated variables is reduced such that the problem becomes arc-consistent. This is achieved by performing one of the arc-consistency algorithms instead of Reduce_Domains in line 7 of FC. In contract to its name, full lookahead does not perform complete possible lookahead, only one assuring arc-consistency. In principle, there are lookahead algorithms possible which achieve higher consistency for the remaining problem. However, such algorithms are not used in practice due to the high time-complexity of the consistency-algorithms.

In-between the forward checking and full lookahead algorithms is **partial lookahead** or directed arc-consistency lookahead, which assures directed arc-consistency for the remaining problem, assuming a fixed ordering of the variables.

4. Structure-based algorithms

According to Theorem 2, if a CSP is directed arc-consistent, then it can be solved without backtracking. As it can be seen easily, for a tree-structured problem directed arc-consistency can be achieved in at most O(nd) steps. Hence tree-structured problems are easy to solve.

The methods to be discussed in this chapter all rely on the above fact, namely that binary tree-structured problems can be solved efficiently. The applicability of the methods require some features of the primal or dual graph of the problem, which can be checked by - sometimes complex - well-known algorithms of graph theory (Even 1979).

4.1. Decomposition into subproblems

If a CSP can be decomposed into independent subproblems - ones which do not share variables -, then obviously these subproblems should be solved independently. There is a fast algorithm to decompose a graph, so it is worth to check the connectivity of the problem before deciding about the solution method. Even if the primal graph of a CSP is connected, there may be a small subset of the variables identified such that removing these variables, the remaining problem can be decomposed into independent subproblems. These independent subproblems can be solved in O(d) steps, where s is the number of variables in the biggest subproblem. Hence the entire problem can be solved in O(d) steps, where r is the number of elements to be removed to make the problem decomposable. The ideal case is such a choice for the elements to be removed when r+s is minimal. Unfortunately, there is no efficient algorithm known to produce such and ideal decomposition of a graph.

4.2. Problems with tree dual graph

The dual problem of any CSP is always a binary CSP. The solution of the dual problem requires that the vertices representing constraints get instantiated, that is, the constraints are satisfied, but in such a way that they "join" along the edges representing shared variables. That is, any two constraints sharing variables must have the same value assigned to the shared variables. If the dual problem is a CSP with tree structure, that is, its primal graph (which is the dual graph of the original CSP) is a tree, then it can be solved easily. Namely, if the original CSP has m constraints, each of them with maximum r solutions, then the dual of the problem can be solved in maximum O(mr) steps. The number of steps with better bookkeeping can be further reduced to O(mlog r). (Dechter 1991).

If the dual graph of a problem is not a tree, it is worth checking if it has redundant edges, and if the removal of these redundant edges reduces the graph to a tree. Particularly, if a set of variables assigned to an edge in the dual graph is such that all the variables are present in the sets assigned to edges along a path connecting the two vertices of the edge in question, then the roundabout path assures that the variables on the selected edge will get matching values. Hence, the edge is redundant and can be removed. The graph gained by removing all redundant edges is the **join graph** of the problem. If the join graph is a tree, then the problem can be solved easily. There are efficient graph algorithms to decide if the dual graph can be reduced to a join tree.

4.4. Decomposition into nonseparable subproblems

The decomposition into nonseparable subproblems method also produces a tree graph, by identifying such subgraphs of the primal graph which, if contracted to a point, the resulting graph is a tree. These subgraphs are the nonseparable components, which can be found in an efficient way. Once they are identified, the original CSP can be turned into one with tree structure, and with some variables the instantiation of which require the solution of the nonseparable components they represent. Such a problem can be solved in O(nd), steps, where r is the size of the maximal subproblem. Hence, this method is worth using if the nonseparable components are all small. However, the worst-case performance is inferior to the one of the adaptive-consistency algorithm. The method can be applied to dual graphs as well.

4.5. Cycle cutset

The cycle cutset method is used for binary CSPs. It identifies nodes the removal of which would turn the constraint graph into a cycle-free one. Such a set of nodes is a **cycle cutset**. Once the variables in a cycle cutset are instantiated in a consistent way, the remaining tree-structured problem can be solved or unsolvability can be proven without backtracking. In the latter case, a new instantiation of the variables in the cycle cutset should be tried. Hence, the tree search algorithm should be invoked at most as many times as the number of solutions of the problem in the cycle cutset. That is, the time complexity of this algorithm is exponential in the size of the cycle cutset. Because the task of finding a minimal cycle cutset is NP-hard, the method is used interwoven with backtrack search, looking for a cycle cutset in an opportunistic way. Namely, by checking if the already instantiated variables are a cycle cutset. If so, the rest of the problem can be solved without backtracking. The tree search assumes directed arc-consistency of the problem, hence the method can be used only if the order of variable instantiations is static.

5. Local methods

Local search methods are also used to solve CSPs. These methods all try to find a solution by stepping from one complete instantiation to another one in its neighbourhood. This is achieved by changing the value of a few - usually one - variables at a time. The algorithms differ in their policy of selecting and changing the variables to be altered. This can be done in a deterministic way, or by incorporating random elements in the choices. The first category contains traditional local search methods applied to CSPs. In the second case, running the algorithm several times it is likely that it will follow different paths and terminate with different results. Hence these algorithms are designed to be used to generate many candidate solutions parallel and/or one after the other, and to keep the best candidate found. We will outline how genetic algorithms as representatives of this category can be applied to CSPs. A third possibility is that on the basis of the evaluation of the effect of the local perturbations, the principle of changes is tuned in a way that finally a solution is produced. Neural networks apply this principle.

Multiple trials, learning and different tricks used with local search are all meant to assure that a solution will be found in spite of the local scope of the search. All the same, completeness cannot be guaranteed in general, neither for deterministic nor for nondeterministic algorithms. However, the random elements do increase the diversity of the search, giving a chance of escaping from mistaken biases of the deterministic decisions. That is why random elements play an essential role in many of the algorithms. Random elements can be added at different points: to generate a "starting instantiation", to decide about which variable to change and to what value. On the other hand, too much random choice degrades the search to random walk and makes it very unlikely to find a solution fast. Hence the art of designing efficient algorithms is in combining heuristics to focus the search with random elements to proliferate it. These algorithms are very sensitive to the structure of the solution space: the amount and distribution of solutions and "almost solutions".

In spite of all these uncertainties, for big and difficult problems which cannot be tackled by constructive search methods, non-deterministic local methods are not only alternatives to be tried, but have proven to be successful in many real-life applications.

In the rest of this chapter we introduce the three types of local methods, and discuss the possibility of coupling constructive and local methods.

5. 1. Local search

The skeleton of the local search methods used to solve CSPs is given below:

```
Procedure Local_Search (variables, domains, constraints)
1. instantiation ¬ (a complete instantiation)
2. stop ¬ FALSE
3 while (not stop) loop
4. x ¬ pick_a_non_compatible_varible(instantiation, constraints)
5. w ¬ pick_a_not_worse_value(instantiation, x, domains,
constraints)
6. instantiation¬ instantiation \ {<x, v>} È {<x, w>}
7. if solution(instantiation, constraints) stop ¬ TRUE endif
8. endloop
9. return (instantiation)
```

```
        end
```

Using the terminology of general local search, in the above algorithm the neighbourhood of the current element - a complete instantiation - consists of all instantiations differing from the current one in at most one variable. The pick_a_not_worse_value procedure assures that at each step the new instantiation generated is not worse than the previous one. The most straightforward way to achieve this is to let the pick_a_not_worse_value procedure a value (randomly selected if there are more equally good ones) such that the new instantiation with this value satisfies at least as many constraints as the current one. But the improvement can be maximised, by selecting the value producing the least constraint violations. This is the principle of the min-conflict heuristic repair method (Minton 1990, Selman 1992).

By local search there is the danger of arriving at a peak or a "plateau", that is, at an element which has only elements in the neighbourhood which do not mean improvement. To avoid such traps, the algorithm should be extended with some explicit stopping criterion and mechanism to force a "jump" out of the current neighbourhood if trapped in a neighbourhood.

5. 2 Genetic algorithms

In the simplest scenario of using a genetic algorithm (GA) to solve a CSP, a member of a population consisting of a given number of complete instantiations is chosen, and a new instantiation (a child) is generated by some perturbation of the chosen one (the parent). Afterwards the child is evaluated and it is decided if it should replace one of the elements in the population. The evaluation is given by a function, the so-called **fitness function**. The algorithm is aiming at producing an element with maximal fitness. It prefers fitter elements both to be selected to be a parent and to survive. However, this is done in a probabilistic way, so poorer elements get a chance too. When generating children, heuristics and random elements both are used.

In case of CSPs, the fitness function can be defined e.g. as the number of satisfied constraints or as the number of "good" variables, that is the ones referred to only by satisfied constraints. As heuristics, the ones used for variable and value selection in constructive search methods can be adapted and used on a probabilistic basis. Already the "one parent - one child" schema allows a variety of genetic algorithms. Most of the applications to solve CSPs use such a GA (Eiben 1997). However - according to the original principle of GAs - a new instantiation can be produced by selecting 2 or even more parents and inheriting some parts from each of them. In such a scenario the structure of the constraint graph and classical heuristics used by graph-based methods can be adapted to design the GA: what number of parents to be considered, how to select promising parents and how to inherit parts of the instantiations by the child to be generated from them. It is also a used practice to refine and tune the fitness function from population to population, as a result of learning about the difficulty of variables and subproblems.

5.3. Neural networks

Neural network-based solution methods work on a network with vertices representing all the possible <variable, value> pairs of the CSP to be solved, and the edges have weights. A vertex is either live (on) or dead (off). The live vertices provide input for all their neighbouring vertices. Each vertex evaluates the weighted effect of the inputs, and preserves its state or changes it to the opposite. This process keeps repeating until an equilibrium state is reached. If the reached state is not acceptable, that is, many constraints are violated, then some of the weights assigned to the edges are changed, which may start another sequence of changes. The structure of the network and the evaluation of the inputs assures that a state always corresponds to a complete evaluation of the variables, and that only conflicting variables have an effect on each other. The strength of the algorithm is in its mechanism to learn, which makes it possible to explore the solution space and to escape from states which are not solutions by adapting weights. Though this algorithm cannot guarantee that a solution will be found, the empirical results are very encouraging (Tsang 1992).

5.4. Combining constructive search and local methods

Initially the researchers of the classical methods and those of the local and non-deterministic methods were competing in trying to show that their own approach is the more successful. Recently this has been replaced by the recognition that these two very different paradigms can well co-exists: none of them provides the winning approach, but when combined, they can compensate each other’s weaknesses, and hence surpass the single-paradigm methods of both kinds (Freuder 1995).

One of the possibilities is to use local methods and constructive search in a sequential way. Local search can be used to investigate the environment of a partial solution and use the evaluation of the exploration as a dynamic heuristics for value selection. Or in a different scenario local search is used to find an "almost solution", which is improved by systematic backtracking search. By using this idea for partial instantiations, many open test CSPs could be solved for the first time (Zhang 1996).

Another possibility is to couple local search with systematic or opportunistic constraint propagation. The famous GSAT local search algorithm could solve difficult problem instances after a certain level of consistency was assured before running the local search (Kask 1995).

6. Comparison of problems and solution methods

We have introduced a number of solution methods, some of which can be even combined with each other. How should one, when confronted with a given CSP, choose from this arsenal of methods? It is not sufficient, possibly even misleading, to consider the worst-case or average complexity of the algorithms - for the given problem a generally slow algorithm may perform well. On what basis can one forecast the performance? What type of methods are most promising for a given problem? What should we understand by the difficulty of problems? Can the difficulty be judged on the basis of the structure and other features of the problems? The answers for these questions are not only useful for practical applications, but also give an insight into the structure of a family of NP-complete problems, which can direct further theoretical research, e.g. to discover new solution methods. The analysis of the difficulty of problems - a flourishing research topic since the mid-90s - has produced some very useful and exciting results.

6.1. Easy, hard and exceptionally hard problems

Intuitively, a problem is easy if it can be solved fast. One would say that the less the proportion of the number of solutions and the size of the solution space is, the more difficult the problem is. If a problem is unsolvable, then in the most difficult cases this can be shown only by checking all the elements of the solution space, while in the easiest case the unsolvability can be seen "at a glance", e.g. by noticing that a certain constraint cannot be satisfied. When adding constraints to an initially easy problem with many solutions, the number of solutions decreases, hence the problem becomes harder. On the other hand, the added constraints make more partial instantiations inconsistent. Hence, it can be earlier detected in course of search that a partial instantiation leads to a dead end. Thus one expects that by adding constraints will gradually make the problem more difficult to a certain point, and afterwards the extra constraints will make the problem easier.

Empirical results for several types of CSPs have supported this hypothesis. E.g. for graph 3-colouring it has turned out that there is, indeed, a critical level of constrainedness, namely graphs with 4.6 edge density are the most difficult to colour (Hogg 1994). For graphs with a smaller edge density it was easy to find one of the several solutions, while for graphs with a higher edge density there was no solution at all, which - because of the many edges - was easy to detect. The most difficult problems were found in the transition phase from the region of solvable problems to the region of unsolvable problems. This is the so-called **phase transition** phenomenon: the solvability of CSP instances of a given type can be expressed in terms of a single parameter representing the constrainedness of the problems, and the solvable and nonsolvable instances are separated by a critical value of this parameter. The hard problems are to be found around the critical parameter, in the transition phase, where the expected number of solutions is 1. The phenomenon of phase transition was detected for all investigated problem types, including practical problems (Gent 1995).

Is the above hypothesis of phase transition true for all types of CSPs? How can the single parameter of difficulty be derived for different problem families? A recent analytical investigation gives answers to these questions, and provides a general framework for the empirical results (Gent 1996). Namely, if M is the expected number of solutions for a certain subset of CSPs of a given type, and the size of the solution space is N, then the number k=1-log represents the constrainedness of the problems. In the region k<1 the problems are underconstrained, while for k>1 the problems are normally overconstrained. The critical region is around k=1, where it is difficult to find out if the problem is solvable or not. For graph 3-colouring problems the empirically detected critical edge density corresponds to k= 0.84.

For certain problem types the expected number of solutions, M can be well estimated, and this estimate can be used to judge the difficulty of a problem at hand. The k value is also used to generate a body of difficult graph-colouring or other binary CSPs to be used as test cases to evaluate solution methods.

Another nice way of profiting from the above result is to use it as a dynamic variable ordering heuristic: that variable is taken next for which the corresponding extended problem is the most difficult, according to the k value of the possible extensions. This heuristic seems to outperform the widely used fail first heuristic. On the other hand, for several widely-used classical heuristics it has been shown that they correspond to the optimisation of k, or an easy to compute estimate of it.

One should not forget that a k value always represents the difficulty of a subset of CSPs of a given type based on the expected number of solutions. Hence k tells how difficult the instances of the set in consideration are, on average. It can happen that as an exception, a hard or even extremely hard problems occur in the easy region, which was observed indeed (Smith 1995).

6.2. Which method to use ?

How to select from the ever increasing choice of the applicable heuristics and solution methods when one has to attack a given CSP? In the recent years, two approaches have been taken to solve this dilemma. Thorough empirical and theoretical analysis of the distinct methods and combination of heuristics provides guidelines for their applicability. On the other hand, there is a new paradigm emerging: instead of committing oneself to a single method, one should go for using different solution methods, one after the other or parallel. There are toolkit-like application-domain specific or general CSP solving environments based on this idea.

As discussed in Section 3, uninformed search can be turned into a specific one by extending it in (some of) the following 4 respects:

(a) order of instantiating the variables;

(b) order of trying out possible values for the variable on turn;

(c) amount of propagation of the effect of a hypothetical or performed instantiation;

(d) strategy of backtracking in case of a dead-end.

The distinct methods can be identified by naming the heuristics used at the different points: FFP-FC-CBJ is thus the method which for a) uses the first-fail principle; for c) it performs forward checking; and for d) it uses conflict-directed backjumping. In place of FFP there could stand MWO (minimal width ordering heuristic), or nothing, indicating that no heuristic is used for variable ordering.

Which combinations of heuristics are to be used? When does the application of heuristics pay off? Is it so that the more heuristics are plugged in, the better the algorithm is? Extensive research is still going on to chart the performance of solution methods with one or more heuristics on a large testbed of binary CSPs with different difficulty and other characteristics, such as tightness and number of constraints, size of the problem (Frost 1994, Tsang 1995). These empirical investigations have proven what one would expect: there is no single best solution method. Intelligent backtracking pays off if the constraint graph is sparse, while it is worth to do forward checking and perform constraint propagation if the constraint graph is dense and the constraints are tight. Considering variable ordering, for certain problems structure-based heuristic is suitable, while for others a domain-based heuristic is the more advantageous. Not withstanding all these, there are two methods which are accepted as "generally good, robust ones": MWO-BM-CBJ for CSPs with sparse and non-homogeneous constraint graph, and FFP-FC-CBJ for CSPs with dense constraint graph and tight constraints.

The delicacy of the problem is well illustrated by the case when, in contrast to the general practice, the application of two heuristics decreased the efficiency of the single-heuristic methods (Posser 1993).

Hence choosing the proper solution method for solving a difficult, large CSP should be done with much care, in the light of the theoretical and empirical results on the strength and weaknesses of the distinct methods. This a priori critical decision, requiring expertise, is not necessary if one can switch to more and more expensive and powerful heuristics if necessary, while searching for a solution. The first promising example of this principle is a solution strategy which automatically switches to a more powerful heuristic if the search has made no progress for some time with the current one (Borrett 1996).

We finish the account on the evaluation of solution methods with a positive and exact result: by analysing the exploration of the search tree, some heuristics can be compared in terms of the number of instantiations and the number of constraint checks. E.g. it has been shown that FC-CBJ is always better than FC alone, but it is not necessarily better than CBJ alone (Kondrak 1995). This first work on the partial ordering of heuristics is a very illuminating and by the research community much appreciated first result.

7. Non-classical constraint satisfaction

In real-life situations often emerge problems which cannot be fully captured by the classical definition of CSP. It is quite normal that not a single CSP has to be solved, but a series of CSPs: because of - usually slight - changes in the circumstances, a solution of the accordingly modified CSP has to be found from time to time. Often it is also required that the solution of the modified problem is close in some sense to the solution provided for the previous one. But there are situations - e.g. exploring possible alternatives for a design - when one would like to get a very different solution by relaxing some constraints. The third most important phenomenon in real problems is that not all the constraints are of equal importance. If not all the constraints can be satisfied, the user would be quite happy with a "good enough" solution, that is with such a complete instantiation for which some constraints are violated, but only those and to such an extent that the violations are still acceptable for the user. Such a compromise may even be necessary for solvable problems too, because of the limited time available to find the complete solution.

CSP researchers have been extending the original concept of the CSP and enhancing the traditional solution methods in a variety of ways, in order to bridge the gap between the needs of real-life applications and the rigour of the classical framework. Below, we discuss some representative approaches.

7.1. Partial constraint satisfaction

From the point of applicability probably the most unpleasant limitation of the classical CSP framework that it cannot do anything other than reporting failure if a problem is unsolvable - rather the rule than the exception in real-life cases. In such a case the user knows that he has to sacrifice some constraints anyway. He would like to know how this can be done the best, that is by relaxing the problem to a nearest solvable one. In parallel to the research on classical CSPs, already in the mid-80s the first steps were made to generalise the CSP in this direction. In case of partial constraint satisfaction, not a single CSP is to be solved, but a partially ordered metric space of CSPs over the same set of variables is considered. The task is to find a solution of a best, or good enough problem, in terms of the given metric.

In case of **partial constraint satisfaction** elements of the metric space - all CSPs - contain an initially given *P* problem and some of its relaxations. The partial ordering reflecting relaxation is defined by comparing the solution sets: $P^{3} Q$ if all the solutions of *P* are also solutions of *Q* , where *P* and *Q* are CSPs defined on the same set of variables. E.g., one can consider the relaxations gained by removing 1, 2, 3, ... constraints from the initial problem. The metric indicates how acceptable the relaxed problems are. On the above introduced set of relaxed problems a straightforward metric is the number of constraints. The higher the metric of the problem is, the better it is. The task of finding the solution of the corresponding partial CSP is to find a solution for a subproblem which contains the maximum number of constraints from the initial problem. (Note that there may be more than one solvable subproblems with the maximum number of constraints.)

If not all the constraints are equally important, this can be reflected by defining the set of CSPs and/or the metric differently. If some constraints cannot be sacrificed, then these are labelled as hard constraints, while the rest are the soft constraints. Then the problem space consists of only those relaxations of the initial problem which are gained by removing 1,2, 3,..., of the soft constraints. This scheme can be further refined if the constraints have different importance, given by the user. It is also him who has to define the metric in terms of the importance of the individual constraints: whether he wants to have the most important constraints to be satisfied as possible, or rather have the total importance of the satisfied constraints maximum. A variant of partial constraint satisfaction is the so-called hierarchical constraint satisfaction (Borning et al. 1992), when the problems in the solution space are evaluated based on the hierarchy of the constraints.

The partial constraint satisfaction framework also allows that a given problem gets relaxed along its variables, by extending the domain of the variables step by step. In this way one can assure that a solution near to a given instantiation - one which is preferred by the user - is to be found.

A similar generalisation is gained by introducing fuzzy constraints in place of the traditional crisp constraints. A traditional crisp constraint can be described by a function which assigns 1 to instantiations for which the constraints hold and 0 otherwise. A **fuzzy constraint** expresses to what extent an instantiation of the variables fulfils the constraint, by assigning to each of the instantiations a value between 0 and 1 (Ruttkay 1994). The degree of satisfaction of more fuzzy constraints also given by a value between 0 and 1, derived from the degree of satisfaction of the individual fuzzy constraints, e.g. as their maximum, average or product. Fuzzy constraints are handy to model qualitative or vague requirements and incomplete knowledge.

For possible extensions of the classical CSP and their applicability, unifying discussion of problem spaces (Bistarelli 1995) and the adaptation of classical search methods is recommended (Freuder 1992).

7.3. Special domains and constraints

In the classical CSP setting the domains are finite and unstructured, which makes the modelling of certain real problems unhandy or impossible. A typical case is when the constraints are to express temporal relations. In principle, a problem with temporal relations can be formulated as a classical CSP. Time has to be discretized, which may result in very large domains. The second problem is that the traditional framework cannot exploit the ordering of the domains and the specific semantics of temporal relations. That is, if a constraint prescribes a lower limit for the time of the event as x ³ 100, and after discretizing time the domain for x is {1,2,3,..., 100, 101, ..., 300}, then in the traditional framework for all the 300 possible values for x the

x ³ 100 will be checked, while by exploiting the fact that the domain of x is an interval of integers, and the ³ is transitive, one would know immediately that the values satisfying the relation are { 100,..., 300 } .

This is happening in the case of **temporal constraint satisfaction** problems. The speciality of the temporal CSPs is that the domains are intervals, and the constraints are from a set of specific, qualitative (like precedence of time of events) and/or quantitative (explicit limits on duration) constraints (Schwalb 1998). The general TCSP is also NP-hard, which is very critical from the point of view of the applications: usually, a problem concerning time, like scheduling, has to be solved within a reasonable time. However, by restricting the types of temporal constraints, the specific TCSP can be solved in polynomial time. Such a sub-class, called the **simple temporal problem** is gained by allowing only constraints giving a single lower or upper limit for time intervals.

A natural generalisation of CSPs comes by lifting the finiteness requirement of the domains. In the case of the **numerical constraint satisfaction problem** (NCSP), the domains are intervals of reals. The domains and the search space can be efficiently represented by using the endpoints of the intervals. The propagation of values of the classical solution scheme is replaced by propagation of intervals. The concept of classical arc-consistency is replaced by certain consistency for the endpoints of the intervals, which can be achieved by algorithms similar to AC-3 (Lhomme 1993). The focusing of the search takes place by splitting the interval in the domain of the current variable into two, and restricting the search to one of them. The process terminates either by showing that the problem has no solution, or by narrowing the interval containing at least one solution until an initially given approximation of the solution is reached. In the last years there is much interest in these methods suited for engineering and scientific applications. Some of the special solution methods have been incorporated into constraint logic programming languages which are (also) prepared to solve NCSPs.

If the constraints are all linear equations over reals, and the task is to find a solution which minimises a linear objective function, then this special COP is the well-known problem of **linear programming**. There are special solution methods like the most common simplex method, to solve linear programs. Methods have been developed to solve the problem in linear time. Though some constraint programming languages do support linear programming, they do not compete with the performance of dedicated linear programming packages. However, the integration of constraint satisfaction and linear programming is useful when a general CSP or COP can be reduced or decomposed to linear problems. For an overview of further possibilities, see (Van Hentenryck 1995).

A specific type of the general CSP is the **satisfiability problem** (SAT). The variables are all Boolean variables with the possible values {true, false}. Each constraint is a disjunctions of literals, where each literal is a variable or a negated variable. The task is to find an assignment of the Boolean variables for which the Boolean expression corresponding to the conjuction of constraints is satisfied. SAT is important because it links CSP to deductive reasoning, and also because it is the reference NP-complete problem. For discussion of tractable sub-classes of and special solution methods for SAT see (Rauzy 1995).

7.4. Generation, modification and analysis of CSPs

In the classical CSP framework CSPs are dealt with in a stand alone way, without providing means to generate a CSP or to exploit the way it was generated. The solution methods were designed to produce a solution, but not to provide analysis of the solution neither to allow interaction with the user in course of the search for a solution. Below, we will discuss how certain generalisations of the classical CSP deal with some of these shortcomings.

A straightforward generalisation of the classical scheme, the so-called **mixed constraint satisfaction** (Fragier 1996) is achieved by differentiating the role of the variables. E.g. in diagnostics, one is interested in the value of the hidden variables of the problem, assuming that the observed variables have certain values. The same cast of role is useful to model control problems. In these situations the existence of "many solution" is just the unfavoured case: from the observations one cannot derive the unique diagnosis, or the chosen setting of control parameters does not define the outcome uniquely. Along similar extension it is possible to handle variables and solutions on different levels of abstractions.

It is often the case that the particular CSP to be considered depends on certain aspects of the environment. Hence, the actual problem to be solved can be generated by analysing variables reflecting the state of the environment. **Dynamic constraint satisfaction** is capable to cope with such situations. Next to the traditional constraints, there are some so-called "variable activation" constraints. With the explicit declaration of these constraints it is possible to model e.g. different stages of an evolving design, or to model scheduling tasks in a way that the problem is updated as a reaction to certain events (e.g. breakdown of machines). Dynamic CSP is also used in a different sense, namely for a sequence of such CSPs such that each CSP is "almost the same" as the preceding one. In this case the main research issue is how to re-use solutions and special characteristics of the lately solved problem.

Finally, a main shortcoming of the classical CSP methods is that they are not enhanced to give an explanation for the derived result. Not all the solution methods are suitable to provide an explanation meaningful for the user, either because of the type or the level of details of the processing. For generating explanation when using a traditional and several, especially **"explanation-aimed" constraint propagation** methods, see (Sqalli 1996).

References

(Bistarelli 1995) S. Bistarelli, U. Montanari, F. Rossi, Constraint solving over semirings, *Proc. of IJCAI*, 1995, 624-630.

(Borning et al. 1992) A. Borning, B. N. Freeman-Benson, M. Wilson, Constraint hierarchies, *Lisp and Symbolic Computation,* Vol. 5. 1992, 223-270.

(Borning et al. 1995) A. Borning, B. N. Freeman-Benson, The OTI constraint Solver: a constraint library for constructing interactive graphical user interfaces, *Proc. of Principles and Practice of Constraint Programming,* 1995, Springer, 624-628.

(Borrett 1996) J. Borrett, E. Tsang, N. Walsh, Adaptive constraint satisfaction: the quickest first principle, *Proc. of ECAI*, 1996, 160-169.

(Bowen 1992) Bowen, J., Bahler, D. Frames, Quantification, perspectives, and negotiation in constraint networks for life-cycle engineering, *Artificial Intelligence in Engineering* Vol. 7. 1992, 199-225.

(Bressiere 1995) C. Bressiere, E. C. Freuder, J. C. Regin, Using inference to reduce arc-consistency computation, *Proc. of IJCAI*, 1995, 592-598.

(COSYTEC 1995) CHIP C library, COSYTECT SA, Orsay Cedex, France

(Dechter 1988) R. Dechter, J. Pearl, Network-based heuristics for constraint satisfaction problems, *Artificial Intelligence* Vol. 34. 1988, 1-38.

(Dechter 1991) R. Dechter, J. Pearl. Directed constraint networks: a relational framework for casual modeling, *Proc. of IJCAI*, 1991.

(Eiben1997) G. Eiben, Zs. Ruttkay, Constraint satisfaction problems and evolutionary algorithms, In: T. B. Bäck, D. Fogel, Z. Michalewicz (eds.), Handbook of Evolutionary Algorithms, Oxford University Press, 1997.

(Even 1979) S. Even, Graph Algorithms, Computer Science Press 1979.

(Fargier 1996) H. Fargier, J. Lang, Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge, *Proc. of AAAI*, 1996, 175-180.

(Fox 1989) M. Fox, N. Sadeh and C. Baykan, Constrained heuristic search, *Proc. of IJCAI*, 1989, 20-25.

(Frayman 1987) F. Frayman, S. Mittal, COSSACK. A constraint-based expert system for configuration tasks, In: D. Sriram, R. A. Adey (ed*.). Knowledge-Based Expert Systems in Engineering: Planning and Design*, Billerica, Mass. Computational Mechanics Publications, 1987.

(Freuder 1978) E. C. Freuder, Synthesizing constraint expressions, Communications of the ACM Vol. 21. 1978, 958-966.

(Freuder 1982) E. C. Freuder, A sufficient condition for backtrack-free search, Journal of the ACM Vol. 29. 1982, 24-32.

(Freuder 1992) E. C. Freuder, R. Wallace, Partial constraint satisfaction, *Artificial Intelligence* Vol. 58. 1992, 21-70.

(Freuder 1995) E. C. Freuder, Systematic versus stochastic constraint satisfaction, Panel discussion, *Proc. of IJCAI*, 1995, 2027-2032.

(Frost 1994) D. Frost, R. Dechter, In search of the best constraint satisfaction search, *Proc. of AAAI*, 1994, 301-306.

(Frost 1995) D. Frost, R. Dechter, Look-ahead value ordering for constraint satisfaction problems, *Proc. of IJCAI*, 1995, 572-577.

(Gent 1995) I. Gent, T. Walsh, Phase transitions from real computational problems, *Proc. of 8th International Symposium on AI*, 1995, 356-364.

(Gent 1996) I. Gent, E. MacIntyre, P. Prosser, T. Walsh, The constrainedness of search, *Proc. of AAAI*, 1996, 246-252.

(Hogg 1994) T. Hogg, C. Williams, The hardest constraint problems: A double phase transition, *Artificial Intelligence* Vol. 69. 1994, 359-377.

(Hooker 1994) J. N. Hooker, H. Yan, Verifying logic circuits by benders decomposition, In: (Saraswat 1994)

(Huffman 1971) D. A. Huffman, Impossible objects as nonsense sentences, In: R. Meltzer, D. Michie (eds.), Machine Intelligence 6, Elsevier, 1971, 295-323.

(ILOG 1995) ILOG SOLVER: Object-oriented constraint programming, http://www.ilog.com/ilog/Products/Solver/Conference/

(Kask 1995) K. Kask, R. Dechter, GAST and local consistency, *Proc. of IJCAI*, 1995, 616-622.

(Kondrak 1995) G. Kondrak, P. van Beek, A theoretical evaluation of selected backtracking algorithms, *Proc. of IJCAI*, 1995, 541-547.

(Lhomme 1993) O. Lhomme, Consistency techniques for numeric CSPs, *Proc. of IJCAI*, 1993, 232-238.

(Mackworth 1977) A. K. Mackworth, Consistency in networks of relations, *Artificial Intelligence* Vol. 8. 1977, 99-118

(Minton 1990) S. Minton, M. Johnston, A. Philips, P, Laird, Solving large-scale constraint satisfaction and scheduling problems using heuristic repair method, *Proc. of AAAI*, 1990, 17-24.

(Morris 1992) P. Morris, On the density of solutions in equilibrium points for the queens problem, *Proc. of the AAAI*, 1992, 428-433.

(Nadel 1991) B. Nadel, J. In, Automobile transmission design as a constraint satisfaction problem: modeling the kinematic level, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* Vol. 5. 1991.

(Prosser 1993) P. Prosser, Domain filtering can degrade intelligent backtracking search, *Proc. of IJCAI*, 1993, 262-267.

(Rauzy 1995) A. Rauzy, Polynomial restrictions of SAT, *Proc. of Principles and Practice of Constraint Programming*, 1995, 515-532.

(Ruttkay 1994) Zs. Ruttkay, Fuzzy constraint satisfaction, *Proc. of 3rd Int. Conf. on Fuzzy Systems*, 1994, 1263-1268.

(Sannella 1994) M. Sannella, The SkyBlue Constraint Solver and its Applications, MIT Press, 1994.

(Saraswat 1994) V. A. Saraswat, P. van Hentenryck (eds.), Principles and Practice of Constraint Programming, MIT Press, 1994.

(Schwalb 1998) E. Schwalb, L. Vila, Temporal constraints: a survey, *Constraints*, 1998, (to appear)

(Selman 1992) B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, *Proc. of AAAI,* 1992, 440-446.

(Smith 1995) B. Smith, S. Grant, Sparse constraint graphs and exceptionally hard problems, *Proc. of IJCAI,* 1995, 646-651.

(Sosic 1991) R. Sosic, J. Gu, 3,000,000 queens in less than one minute, *SIGART Bulletin,* Vol. 2. 1991, 22-24.

(Sqalli 1996) M. Sqalli, E. C. Freuder, Inference-based constraint satisfaction supports explanation, *Proc. of AAAI,* 1996, 318-325.

(Sutherland 1963) I. Sutherland, A man machine graphical communication system, PhD Thesis, MIT 1963.

(Tsang 1992) E. Tsang, C. Wang, A generic neural network approach for constraint satisfaction problems, In: J. Taylor (ed.), Neural Network Applications, Springer-Verlag 1992.

(Tsang 1993) E. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.

(Tsang 1995) E. Tsang, J. Borrett, A. Kwan, An attempt to map the performance of a range of algorithm and heuristic combinations, *Proc. of AI and Simulated Behaviour,* 1995, 203-215.

(Van Hentenryck 1992) P. van Hentenryck, Y. Deville, Choh-Man Teng, A generic arc-consistency algorithm and its specializations, *Artificial Intelligence* Vol. 57. 1992, 291-321.

(Van Hentenryck 1995) P. van Hentenryck, Constraint solving for combinatorial search problems: a tutorial, *Proc. of Principles and Practice of Constraint Programming*, 1995, 564-587.

(Wallace 1993) R. Wallace, Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs, *Proc. of IJCAI,* 1993, 239-245.

(Wallace 1996) R. Wallace, Practical applications of constraint programming, Constraints, Vol. 1. 1996, 139-168.

(Waltz 1975) D. Waltz, Understanding line drawings of scenes with shadows, In: P. H. Winston (ed.), The Psychology of Computer Vision, McGraw Hill, 1975.

(Zhang 1996) J. Zhang, H., Zhang, Combining local search and backtracking techniques for constraint satisfaction, *Proc. of AAAI,* 1996, 369-374.

(Zweben 1994) M. Zweben, M. Fox (eds.), Intelligent Scheduling, Morgan Kaufman 1994.

Further resources

International Journal of Constraints (Kluwer), http://www.kluwer.nl

International Journal of Artificial Intelligence (Elsevier)

Vol. **58**. nr. 1-2 is a special issue on Constraint-Based Reasoning,

Journal of Artificial Intelligence Research (only in electronical format)

http://www.jair.org

A "Constraints Archive" site:

http://www.cs.unh.edu/ccc/archive/

Newsgroup: comp.constraints