Artificial Intelligence Review **13:** 215–227, 1999.
© 1999 *Kluwer Academic Publishers. Printed in the Netherlands.*

A Glimpse of Constraint Satisfaction

EDWARD TSANG
Department of Computer Science, University of Essex, Colchester, Essex CO4 3SQ, UK
E-mail: edward@essex.ac.uk; URL: http://cswww.essex.ac.uk/CSP/

Abstract. Constraint satisfaction has become an important field in computer science. This technology is embedded in millions of pounds of software used by major companies. Many researchers or software engineers in the industry could have benefited from using constraint technology without realizing it. The aim of this paper is to promote constraint technology by providing readers with a fairly quick introduction to this field. The approach here is to use the well known 8-queens problem to illustrate the basic techniques in constraint satisfaction (without going into great details), and leave interested readers with pointers to further study this field.

Keywords: 8-queens problem, constraint satisfaction, search

1. Introduction

Constraint satisfaction problems (CSPs) appear in many areas of computer science, especially artificial intelligence. Following the foot steps of disciplines such as robotics and expert systems, constraint technology has come out of the laboratories and gone into real world applications (see below). Constraint technology has been used or looked at by British Telecom, British Airway, French Railway, Cathay Pacific, Port of Singapore and many other organizations. Constraint-based software has become a multi-million Pounds industry. Many researchers or software engineers in the industry could have benefited from using constraint technology without necessarily realizing it. This article aims to promote *constraint satisfaction* technology by explaining what it is about in simple relatively terms. By using the "8-queens" problem as an example I shall attempt to (a) show that there are many ways to solve the same problem, and (b) introduce some of the basic techniques that can be used. This should help readers to decide whether they should look further into this technology.

1.1. *What is constraint satisfaction?*

A constraint satisfaction problem is a problem where one has to find a value for a (finite) set of variables satisfying a (finite) set of constraints (Freuder and Mackworth 1994) (Mackworth 1997) (Tsang 1993). Research in this field involves finding methods to solve such problems efficiently.

Constraints can be found in many places in daily life: regulations, restrictions, requirements, machine capacity and preferences are all constraints. One major application is scheduling. For example, airline companies have to schedule crews to flights, and meet aviation regulations and company requirements. Staff rostering in hospitals must satisfy restrictions on team composition, personnel regulations and perhaps preferences. In schools, timetables must be generated in such a way that no teacher will teach two different classes at the same time in two different places. I shall use the famous “8-queens” problem here to introduce some of the basic techniques used in constraint satisfaction.

1.2. *What is the 8-queens problems?*

The 8-queens problem is a well known puzzle among computer scientists. The problem is to place eight queens on eight different squares on a chess board (which has eight rows and eight columns), satisfying the constraint that no two queens can threaten each other. A queen can threaten any other pieces on the same row, column or diagonal. Figure 1 shows one of the many solutions to the 8-queens problem.

Why should anyone be interested in the 8-queens problem? While being an interesting intellectual challenge to some, the 8-queens problem does not resemble any real life problems. The only reason for using this problem here is *simplicity* – this problem is simple to describe but sufficiently difficult to require the techniques that we want to illustrate here. Most real life problems need a lot of time to explain, and many details need to be remembered in order to follow the discussion.

1.3. *How to solve the 8-queens problem?*

There are two basic classes of strategy for this (or any other) constraint satisfaction problem:

(1) *systematic search strategies* – put one queen onto the board at a time and make sure that no constraint is violated, until all eight queens are placed. If at any point one cannot find a safe place for a queen, remove the queen just placed (this is called *backtracking*), and place it in an alternative position which has not been tried. If the squares are tried systematically, all possible board situations will be tried if necessary.

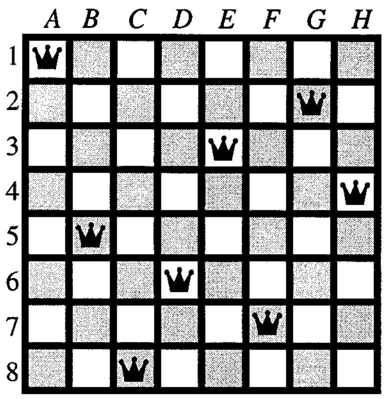

Figure 1. One solution to the 8-queens problem – no two queens are on the same row, same column or the same diagonal.

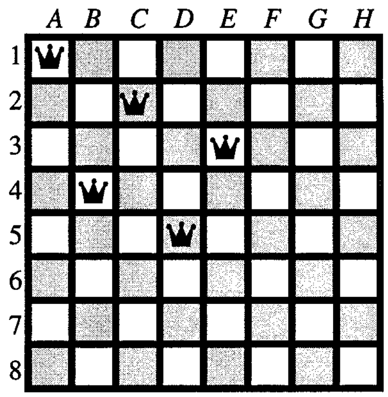

Figure 2. No legal space is available in row 6.

(2) *repair strategies* – put all eight queens onto the board initially at random; then, if any queen threatens another, try to move it to a new place. The hope is that solutions can eventually be found through such *repairs*.

2. Systematic Search Strategies

2.1. *Backtracking search, a naive strategy*

A simple backtracking strategy for solving the 8-queens problem can be performed in the following way: rows 1 to 8 are looked at one at a time in numerical order. For each row, the columns A to H are looked at one at a time, from left to right, and a queen is placed in the first empty space which is not in conflict with any of the queens placed (in the rows above) so far. If all the spaces in the current row are illegal, then remove the previous queen and attempt to place it in an alternative column. Backtrack again if necessary.

For example, when simple backtracking is applied, the first five queens will be placed in the board shown in Figure 2. Then it is found that no legal space is available in row 6. In this case, the preceding queen, 5D, is removed. The queen in row 5 will be placed in the next legal space, which is 5H. Then the search will proceed to row 6 again. If it is found later that 5H leads to dead-ends too, then 4B will be repositioned, and so on.

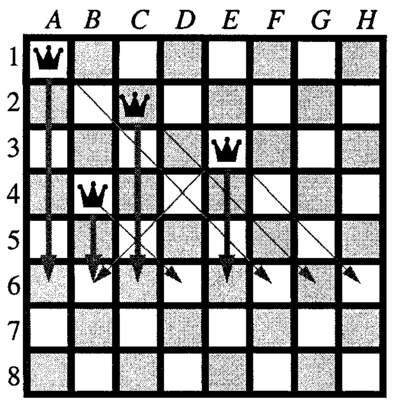

Figure 3. After four queens are placed, no legal space is available in row 6.

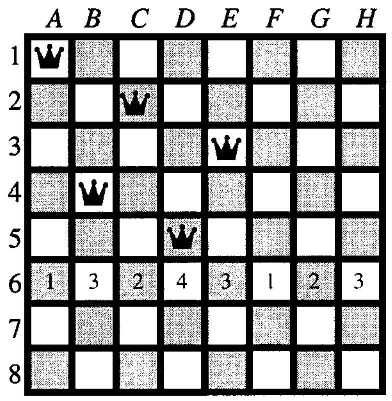

Figure 4. Recognizing culprits: the earliest decision which has ruled out each square in row 6 are marked.

2.2. *Lookahead so as to recognize dead-ends*

If one examines the situation carefully after putting each queen onto the board, one may be able to detect impending dead-ends early. For example, after putting in the first four queens in Figure 2, it is possible to deduce that there is no place to put a queen in row 6: Figure 3 shows that each square in row 6 is attacked by at least one of the first four queens placed. Being able to recognize this is useful, because one can then backtrack before trying to place the fifth queen in 5D and 5H.

In fact with the investment of more effort after putting each queen onto the board, one should be able to deduce that no solution exists after the first three queens are placed in Figure 2 (I shall not elaborate the logic here).

Such *lookahead* is a commonly used technique in constraint satisfaction (Dechter and Pearl 1988). The point is that the effort spent in analysing the situation after each step has to be balanced against the potential gain in avoiding futile search.

One of the simplest forms of lookahead is to invalidate all the threatened squares in the rows still without a queen. This is known as the *forward checking* strategy (Haralick and Elliott 1980). The gain in this example would, in fact, be small, but there are many other situations where the gain can be enormous.

2.3. *Back-jumping and learning from failure*

Another well known systematic search strategy is *back-jumping* (Haralick and Elliott 1980) (Prosser 1993). There are several variations and here is a simple one.

Suppose one does not perform lookahead but when dead-end situations such as the one described in Figure 2 are encountered, one attempts to find out what caused the dead-end. In Figure 4, the earliest decision which has ruled out each of the squares in row 6 is marked. For example, the earliest queen that attacks 6C is the one in row 2, although the queen in 5D attacks it as well. The listed decisions reveal that the latest culprit is the decision made for row 4. Therefore, one can ignore the alternative positions in row 5 and undo 4B immediately.

Note that the back-jumping and the forward checking strategies described above both achieve the same effect: alternatives in row 5 are ignored. More sophisticated back-jumping and lookahead strategies can achieve different effects. It is also possible to combine lookahead and jumping back strategies, though this will require more book-keeping.

The general principle behind back-jumping is to analyse the situation at dead-end situations to find out what the culprits are, and undo the latest one. The alternatives for the queens between that culprit and the dead-end are ignored.

This principle can be pushed even further: by analysing dead-ends more carefully, one can discover "*no-goods*", combinations of decisions which can be rejected whenever they are encountered again (Prosser 1993) (Richards et al. 1995). For example, a little reflection should convince the readers that 1A, 2C, 3E and 5D together can also cause a dead-end in row 6. Therefore, when 4B is replaced by 4G or 4H later, 5D can be rejected immediately. In some cases, such savings can be very large.

2.4. *Ordering in placing the queens*

So far, we have assumed that the queens are placed from row 1 to row 8, and for each row, the columns are considered from A to H. In fact, these orderings can significantly affect the efficiency of a search.

One strategy, when applied to the 8-queens problem, is to place a queen in the row which has the least number of choices next. This strategy has been found to work well with many constraint satisfaction problems. It is sometimes referred to as the *fail-first principle* (though recently it has been proved to be an inappropriate name; interested readers should refer to (Grant 1988) for details). This strategy works well with the forward checking strategy described above. Assume that the first three queens have been placed in rows

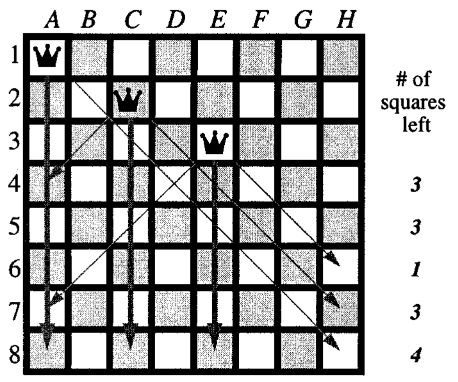

Figure 5. The numbers on the right indicate the number of safe squares left in that row, Row 6 has only one safe square left.

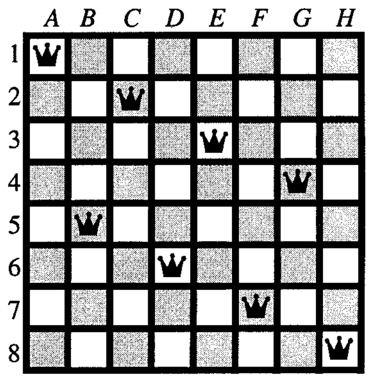

Figure 6. Repair method: 1A and 8H attack each other; if 1A is chosen for repositioning, squares 1A, 1E and 1H are better than the others as they are each attacked by one queen only.

1 to 3, and all the conflicting squares have been removed in the empty rows, as shown in Figure 5.

The number of available squares left in each of the empty rows is shown in Figure 5. The most constrained row is row 6, which has only one square unattacked, namely, 6D. A sensible decision is to put the fourth queen in 6D (rather than 4B). In this example, we are putting a queen in a row with no alternatives, which is obviously sensible. The point is: even if the most constrained row has more than one square available, it is still a good idea to place a queen in this row next.

Similarly, the order in which the available squares in a row are tried is significant for the efficiency of a search. One idea is to pick the most promising square first.

3. Repair Strategies

All the above methods assume that one queen is placed in the board at a time, and backtrack when necessary. Another approach is to start with 8 queens on the board, which may attack each other, and keep repositioning those illegal queens (in other words, ‘repairing the board’) until a solution is found or one runs out of patience. Such repair strategies are often called *heuristic search* or *stochastic* strategies (Reeves 1991).

3.1. *A simple repair method*

In this section, we shall introduce a simple repair method called the *min-conflict heuristic repair method* (Minton et al. 1992). The starting point can be obtained by putting 8 queens in arbitrary positions in each of the 8 rows. Alternatively, one can attempt to engineer a board situation with as few conflicts as one can conveniently get. Figure 6 shows a situation which was created by putting one queen at a time from row 1 to row 8, minimizing the number of attacks for each row. For example, a queen is placed in 7F since it is attacked by no other queens above it. A queen can be placed in 8A or 8H because they are both attacked by only one other queen; in Figure 6, 8H is chosen arbitrarily.

In Figure 6, the only queens that attach each other are in 1A and 8H. One repair strategy is to pick one of them at random, and attempt to reposition it in the same row, in a square that is attacked by the least number of queens, breaking ties randomly. Actually randomness is found to play an important role in repair methods.

Assume that in Figure 6, 1A is picked for repositioning. A count would reveal that 1A, 1E and 1H are each attacked by one other queen only, while all the other squares in row 1 are attacked by two or more queens. There is nothing to stop 1A being picked if a random choice is to be made, but to make it more interesting, let us assume that 1E has been picked. The board situation after this “repair” is shown in Figure 7.

In Figure 7, 1E and 3E are the only two attacked queens. One of them will be picked randomly for repair. If 3E is picked, then 3A and 3E are better choices, as they are each attacked by one other queen, while all the other squares are attacked by two or more queens. The solution shown in Figure 8 will be found if one by any chance moves 3E to 3A, 6D to 6H, and then 8H to 8D.

3.2. *Alternative repair strategies*

Based on the observation that in every solution each column must be occupied by one queen, an alternative repair method is to swap the columns of two queens in each repair. In the example in Figure 6, had one chosen to move 1A to 1E, 3E will be moved to 3A at the same time. Similarly, the reposition of 6D by 6H and 8H by 8D will be done in one repair iteration rather than two. This strategy exploits certain properties of the 8-queens problem and therefore is less general than the above repair strategy.

Other ways of repairing a candidate solution have been proposed. For example, GSAT re-starts from new starting points periodically (Selman et al. 1993, 1994) (Gent and Walsh 1993). *Tabu search* imposes restrictions on

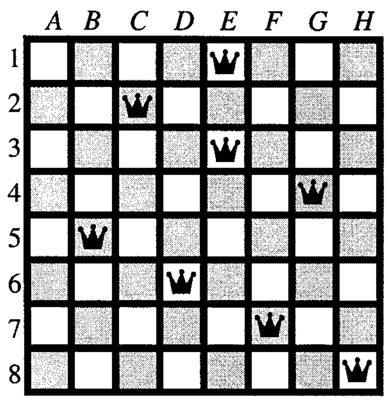

Figure 7. The queens in 1E and 3E attack each other. If 3E is picked for repositioning, then 3A and 3E are the squares attacked by the least number of queens.

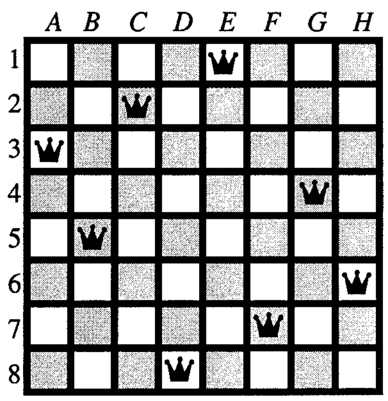

Figure 8. A solution which could be found by repairing the board in Figure 7 in the following way: move 3E to 3A, 6D to 6H, and then 8H to 8D.

repairs that one is allowed to make at any time (Glover et al. 1989, 1993). *Simulated annealing* allows poor moves (i.e. moves which may increase the number of attacks) under certain circumstances (Aarts and Korst 1989) (Chew et al. 1992). Borrowing their ideas from nature, genetic algorithms maintain and manipulate a set (called ‘population’) of candidate solutions (Eiben et al. 1994) (Ruttkay et al. 1995) (Warwirk and Tsang 1995) (Lau and Tsang 1997). *GENET* (Davenport et al. 1994) and *guided local search* (Voudouris and Tsang 1996, 1997, 1998) “learn” about bad moves or bad combinations of positions. These are all interesting strategies, but detailed description of which is beyond the scope of this paper.

4. Using Constraint Satisfaction to Solve Problems

In this section, we ask the question of how to apply constraint technology. Very little research has been done on the software engineering aspect of constraint satisfaction, i.e. given a problem, how should one approach the problem and, if appropriate, apply constraint techniques to solve it. Initial steps have been made in the CHIC (http://www.ecrc.de/CHIC) and Computer-aided Constraint Programming (CACP; http://cswww.essex.ac.uk/CSP/cacp/) projects.

Figure 9 shows a broad outline of applying constraint satisfaction techniques to solve problems. Given a problem description, one way to determine whether it can be solved by constraint satisfaction is to attempt to define three

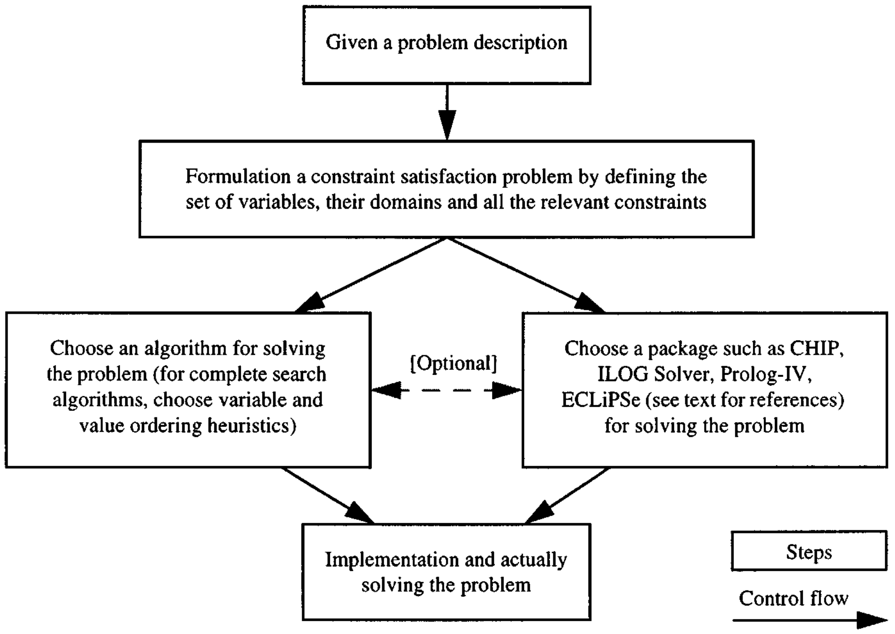
Given a problem description
Formulation a constraint satisfaction problem by defining the set of variables, their domains and all the relevant constraints
Choose an algorithm for solving the problem (for complete search algorithms, choose variable and value ordering heuristics)
[Optional]
Choose a package such as CHIP, ILOG Solver, Prolog-IV, ECLiPSe (see text for references) for solving the problem
Implementation and actually solving the problem
Steps
Control flow

Figure 9. A broad outline of applying constraint satisfaction techniques to solve problems

components: (i) variables, (ii) domains and (iii) constraints. If one can define these components, then one can apply constraint satisfaction techniques to solve the problem. In Section 2, we defined for the 8-queens problem 8 variables, each representing the queen in one row. Each variable is allowed to take values A, B, ..., H. This set of values is called the *domain* of the variables. Assigning value H to variable 4, says, represents placing a queen on row 4, column H. Note that in this problem, all variables have the same domain. In other problems, this may not be the case. We can think of the constraints in this problem as a function: given any pair of assignments, it returns 'violated' if the assignments represent two squares in the same column or the same diagonals; it returns 'satisfied' otherwise.

It is important to note that there are likely to be many different ways to formulate the same problem as a constraint satisfaction problem: by defining the set of variables, their domains and constraints differently. Some formulations could make the problem significantly easier (or more difficult) to solve than others. Little progress has been made in making problem formulation a mechanical procedure. Some preliminary steps have been made by Nadel (who presented many alternative ways to formulate the 8-queens problem as a constraint satisfaction problem (Naded 1990)) and Borrett (who extended Nadel's work (Borrett 1998)).

After formulating a constraint satisfaction, the next task is to find a way to solve it. One possibility is to choose and implement one of the algorithms described above. (Exactly which algorithm to choose is a non-trivial issue, which will be discussed in the next section.) Details of the algorithms described above can be found in Tsang (1993). An alternative is to use packages such as ILOG Solver (Puget 1995), CHIP (Simonis 1995), ECLiPSe (Lever et al. 1995) and Prolog IV (Colmerauer 1990). These packages, which are all results of sound constraint satisfaction research, have been applied to real life problems (see, for example, (Cras 1993) (Wallace 1996) (Zweben and Fox 1994)). They have built-in procedures for solving constraint satisfaction problems, though expert users may implement their own constraints solving algorithm if they want to.

Two points should be noted when considering using commercial packages. Firstly, some users may find the learning curve in certain packages steep. Secondly, the choice of variables, domains and constraints may be crucial to the efficiency, so knowledge about problem formulation and the algorithms used by these packages is still very important. To alleviate these problems, consultancy is available (at extra charge) with most of these packages.

5. Choice of Algorithms

In the previous sections, we have seen the principles behind the main techniques in constraint satisfaction. One important decision is the choice between systematic search and stochastic search, because they have very different approaches and different resource requirements.

In systematic search, one can ensure that all possibilities are tried and therefore solutions will be found if they exist. In a repairing approach, it is not that easy to ensure that all possibilities are exhausted. What one gains in the repairing approach is that solutions can be found more quickly in certain types of problems.

The 8-queens problem is relatively small and most people could find a solution without the aid of a computer. However there are many real life problems that are much larger and would take many months or years to solve without the aid of a computer. Imagine trying to find a solution to the 1,000,000 queens problem. The min-conflict heuristic repair method mentioned above can find a solution within minutes on today’s computers. Systematic search would probably take days.

On the other hand, almost all existing repair methods cannot tell when a problem has no solutions. They can only keep on searching until resources run out. An interesting debate on whether systematic or repair-based methods are more promising can be found in (Freuder et al. 1995). It is reasonable to believe that systematic methods are better for some problems and repair-based methods are better for others.

Under each search paradigm, there are still many search algorithms to choose from. Besides, one may choose heuristics for ordering the variables and values in systematic search. We introduced a heuristic (namely the *fail-first principle*) for ordering the variables in a systematic search in Section 2.4. Many other heuristics have been proposed in the literature. Their performance often vary when worked with different search algorithms. The modern view is that different algorithms and heuristics are suitable for different problems. Unfortunately, very little work has been done in positioning the algorithms and heuristics among constraint satisfaction problems (readers may refer to (Tsang et al. 1995) for preliminary work). This means given a constraint satisfaction, it is extremely difficult to know which algorithm and heuristic are the most efficient for solving it. Expert knowledge is needed. This software crisis (the crisis that expert resources are the bottle-neck in engineering constraint-based software) partly explains why constraint satisfaction has not been more popular, despite its wide applicability and its maturity in algorithms design.

6. Where To Go From Here?

This paper only describes some of the basic techniques in constraint satisfaction. It is not meant to be a survey. Many interesting techniques in this field have not been covered here. Rich and Knight (1991) gives a good (though somewhat dated) account of techniques developed in this field. Tsang (1993) provides a rigorous and thorough account of fundamental techniques. The newer ideas can only be found in research papers. One of the best places to start searching is the Constraints Archives (see References). The journal constraints report some of the frontier research in this field. Research papers occupy a significant proportion of conferences such as *International Joint Conference on Artificial Intelligence*, (American) *National Conference on Artificial Intelligence, European Conference on Artificial Intelligence* and *Principles and Practice of Constraint Programming* (conference).

Acknowledgements

The author is grateful to Paul Chernett, Sam Steel, Michelle Straw and Gillian Kearney for their help in improving the readability of this paper.

References

Aarts, E. & Korst, J. (1989). *Simulated Annealing and Boltzmann Machines*. John Wiley & Sons.

Borrett, J. (1998). *Formulation Selection for Constraint Satisfaction Problems: A Heuristic Approach*, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK.

Chew, T-L., David, J-M., Nguyen, A. & Tourbier, Y. (1992). Sovling Constraint Satisfaction Problems with Simulated Annealing: The Car Sequencing Problem Revisited, *Proceedings, International Workshop on Expert Systems and Their Applications*, 405–416. France: Avignon.

Colmerauer, A. (July 1990). An introduction to Prolog III. *CACM* **33**(7): 69–90.

Constraints Archives: *http://www.cirl.uoregon.edu/constraints/* and *http://www.cs.unh.edu/ccc/archive*.

Cras, J-Y. (1993). *A Review of Industrial Constraint Solving Tools, AI Perspective Series.* AI Intelligence: Oxford, UK.

Davenport, A., Tsang, E. P. K., Wang, C. J. & Zhu, K. (1994). GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. Proc., *12th National Conference for Artificial Intelligence (AAAI)*, 325–330.

Dechter, R. & Pearl, J. (1988). Network-Based Heuristics for Constraint-Satisfaction Problems. *Artifical Intelligence* **34**: 1–38.

Eiben, A. E., Raue, P–E. & Ruttkay, Zs. (1994). Solving Constraint Satisfaction Problems Using Genetic Algorithms. *Proc., IEEE World Confernece on Computational Intelligence, 1st IEEE Conference on Evolutionary Computation*, 543–547.

Freuder, E. C. & Mackworth, A. (ed.) (1994). *Constraint-Based Reasoning*. MIT Press.

Freuder, E. C., Dechter, R., Ginsberg, M., Selman, B. & Tsang, E. (August 1995). Systematic Versus Stochastic Constraint Satisfaction. Panel Paper. In Mellish, C. (ed.) *Proc., 14th International Joint Conference on AI*, 2027–2032. Canada: Montreal.

Gent, I. P. & Walsh, T. (1993). An Empirical Analysis of Search in GSAT. *Journal of Artificial Intelligence Research* **1**: 47–59.

Glover, F. (1989). Tabu Search Part 1. *Operations Research Society of America (ORSA) Journal on Computing* **1**: 109–206.

Glover, F. & Laguna, M. (1993). Tabu Search. In Reeves, C. (ed.) *Modern Heuristic Techniques for Combinational Problems*, 71–141. Blackwell Scientific Publishing.

Grant, S. A. (1998). *Phase Transition Behavior in Constraint Satisfaction Problems*. PhD Thesis, School of Computer Studies, University of Leeds, UK.

Haralick, R. M. & Elliott, G. L. (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* **14**: 263–313.

Lau, T. L. & Tsang, E. P. K. (December 1997). Solving the Processor Configuration Problem with a Mutation-Based Genetic Algorithm. *International Journal on Artificial Intelligence Tools (IJAIT), World Scientific* **6**(4): 567–585.

Lever, J., Wallace, M. & Richards, B. (1995). Constraint Logic Programming for Scheduling and Planning. *Britisch Telecom Technology Journal* **13**(1): 73–80. Ipswich: Martlesham Heath.

Mackworth, A. K. (1977). Consistency in Networks of Relations. *Artificial Intelligence* **8**(1): 99–118.

Minton, S., Jonston, M., Philips, A. B. & Laird, P. (1992). Minizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* 58(1–3) (Special Volume on Constraint Based Reasoning): 161–205.

Nadel, B. A. (1990). Representation Selection for Constraint Satisfaction: A Case Study Using N-Queens, *IEEE Expert* **5**: 16–23.

Prosser, P. (1993). Hybrid Algorithms for the Constraint Satisfaction Problem. *Coputational Intelligence* **9**(3): 268–299.

Puget, J.-F. (1995). Applications of Constraint Programming In Montanari, U. & Rossi, F. (ed.) *Proceedings, Principles and Practice of Constraint Programming (CP '95), Lecture Notes in Computer Science*, 647–650. Springer Verlag: Berlin/Heidelberg/New York.

Reeves, C. R. (ed.). (1993). *Modern Heuristic Techniques for Combinational Problems.* Blackwell Scientific Publishing.

Rich, E. & Knight, K. (1991). *Artifical Intelligence* (2nd edn.), 88–94. McGraw Hill, Inc.

Richards, T., Jiang, Y. & Richards, B. (1995). Ng-Backmarking – an Algorithm for Constraint Satisfaction. *British Telcom Technology Journal* **13**(1): 102–109. Ipswich, UK: Martlesham Heath.

Ruttkay, Zs., Eiben, A. E. & Raue, P. E. (1995). Improving the Performances of GAs on a GA-hard CSP. *Proceedings, CP95 Workshop on Studying and Solving Really Hard Problems*, 157–171.

Selman, B. & Kautz, H. (1993). Domain-Independent Extenssions to GSAT: Solving Large Structured Satisfiability Problems. *Proc., 13th International Joint Conference on AI*, 290–295.

Selman, B., Kautz, H. A. & Cohen, B. (1994). Noise Strategies for Improving Local Search. *Proc., 12th National Conference for Artificial Intellignece (AAAI)*, 337–343.

Simonis, H. (1995). The CHIP System and Its Applications. In Montanari, U. & Rossi, F. (ed.) *Proceedings, Principles and Practice of Constraint Programming (CP'95), Lecture Notes in Computer Science*. Springer Verlag: Berlin/Heidelberg/New York, 643–646.

Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction*. Academic Press: London and San Diego (see *http://scwww.essex.au.uk/CSP/edward/FCS/html* for availability).

Tsang, E. P. K., Borrett, J. E. & Kwan, A. C. M. (April, 1995). An Attempt to Map the Performance of a Range of Algorithm and Heuristic Combinations. *Proceedings, Artificial Intelligence and Simulated Behavior Conference*, 203–216.

Tsang, E. P. K. & Voudouris, C. (March, 1997). Fast Local Search and Quided Local Search and Their Application to British Telecom's Workforce Scheduling Problem. *Operations Research Letters* **20**(3): 119–127. Amsterdam: Elsevier Science Publishers.

Voudouris, C. & Tsang, E. P. K. (1996). Partial Constraint Satisfaction Problems and Guided Local Search. *Proc., Practical Application of Constraint Technology (PACT'96)*, London, 337–356.

Voudouris, C. & Tsang, E. P. K. (November 1998). Guided Local Search and Its Application to the Traveling Salesman Problem. *European Journal of Operational Research* **113**(2): 80–110.

Warwick, T. & Tsang, E. P. K. (1995). Tackling Car Sequencing Problems Using a Generic Genetic Algorithm. *Evolutionary Computation* **3**(3): 267–298.

Zweben, M. & Fox, M. S. (ed.) (1949). *Intelligent Scheduling*. San Francisco: Morgan Kaufmann.