Joint NCC & IIS Bull., Comp. Science, 16 (2001), 45–55
© 2001 NCC Publisher

Solving CSPs with predominating constraints of the “not equal” type

M. Yu. Loenko

In this paper we present a new stochastic search algorithm which is designed to solve finite binary constraint satisfaction problems, where constraints of the “not equal” type predominate. The experiments show that, in comparison with the backtrack-based algorithms, the presented algorithm is superior when it is used to solve problems of large dimension with many solutions.

1. Introduction

A finite constraint satisfaction problem (finite CSP) is the main subject of our consideration. The finite CSP is determined as a set of variables, each of which is associated with a finite domain, and a set of constraints that restrict the values simultaneously taken by the variables. The problems of scheduling, assignment of resources, and many other practical problems can be formulated as CSPs.

In order to solve finite CSPs, a combination of search algorithms [2, 3, 6] and narrowing algorithms (also called constraint propagation algorithms [4]) is used. The latter ones narrow the search space by removing so-called redundant values from it. Not all constraints, however, can be efficiently propagated.

Consider the following example. Suppose that there exists a variable a with a domain $D_a = \{1, \dots, 100\}$ and a variable b with a domain $D_b = \{3, 7\}$. Then we propagate the constraint $a = 2b$ by removing all values except 6 and 14 from the domain of the variable a. The propagation of a constraint, such as $a = 2b$, has the following property: the less elements in the b-domain, the less elements in the a-domain after the removal of redundant values.

Now, let us consider the constraint $a \neq 2b$. In this case, everything is different. If the b-domain contains at least two elements b_1 and b_2 such that the a-domain contains elements $2b_1$ and $2b_2$, the a-domain cannot be narrowed down. And even if the b-domain has the only element b_1, we cannot essentially narrow a-domain down: we can only remove the element $2b_1$ from its domain if it is present there. Therefore, the use of the technique of constraint propagation in the problems with a great number of constraints of the “not equal” type is inefficient.

The most well known problems containing only inequality constraints are the N-queens problem and the map (graph) coloring problem. In the N-queens problem, it is necessary to place N queens on an $N \times N$ chessboard so that no queen is under direct attack from any other one. The problem of coloring is in assigning a certain number (color) to each node of a specified graph in such a way that the adjacent nodes are of different colors. These problems can be scaled well: by changing the size of the chessboard or the graph, we can make the problem simpler or more difficult. Therefore, they are widely used to estimate the quality of various methods for solving CSPs.

Resource allocation problems also have many constraints of the "not equal" type. Thus, if there is a set of jobs that use a resource, then the constraint of non-simultaneity, which is a constraint of the "not equal" type, is imposed on them. If the size of such problems is great, solving them with the help of backtracking [2, 6] and constraint propagation [4] may take several years. Recently in contrast to this, experienced dispatchers, who had not a definite strategy but only subjective estimates and intuition and rather limited number of operations per second, created timetables for big universities manually (this took several weeks).

An algorithm of non-return search (NRS-algorithm), which, in a way, models the manual search for a solution, is presented in this paper.

The paper is structured as follows. Some existing search algorithms are described in Section 2. Then, a mechanism of manual scheduling is considered. The NRS-algorithm is constructed on the basis of this mechanism. After that, the algorithm itself is presented. And, finally, the results of some experiments and comparisons are presented.

2. Search algorithms

Thus, a major search algorithm for finite CSPs is backtracking [2, 6]. It is also known as chronological backtracking. This algorithm consists in successive assigning values to variables. Initially, all variables are unassigned. An unassigned variable is chosen at each step. After that, a consistent value from its domain is sought for. The consistent value is such that, together with the values of the assigned variables, it does not violate any constraint on these variables. If the domain of the chosen variable does not contain consistent values, the value assigned to the previous variable is considered incompatible, and return to the previous step is performed. Otherwise, the consistent value found is assigned to the chosen variable. After that, transition to the next step takes place. At this step, another variable is chosen and assigned. If all variables turned out to be assigned, a solution is found.

If all values of the first chosen variable turned out to be incompatible, the problem has no solutions.

There are many different implementations of the backtracking with different strategies of choice:

- choice of the next variable to look at;
- choice of the next value to look at;
- choice of the next constraint to examine.

A successful choice may narrow down the search space.

There exist a great number of modifications of the backtracking. Let us consider some of them.

The backchecking algorithm [3] was developed for applications in which the value consistency test takes much time. If in the consideration of some variable y, its value b turns out to be incompatible with some value a of an assigned variable x, the backchecking memorizes this and does not check the value b until a remains a value of the variable x.

The backmarking algorithm [3] is an improvement of the backchecking algorithm. Similarly to the backchecking, it decreases the number of tests of compatibility by memorizing what values are incompatible with the assigned variables. Moreover, it memorizes the values compatible with the assigned variables, and does not check them either.

Suppose that a consistent value for a variable x_j is sought for at some step. Suppose also that some time ago all possible values of x_j were tested and found incompatible. Assume that since then the backmarking returned to some x_i, where $i < j$, and gave it a new value. After that, it found consistent values for the variables $x_{i+1}, \dots, x_{j-1}$. Then, at the search for a consistent value for the variable x_j , the backmarking will not consider those values that are in conflict with the variables $x_1, \dots, x_{i-1}$; it is already known that they are incompatible. Besides, the backmarking will not check compatibility between $x_1, \dots, x_{i-1}$ and those remaining values that are known to be compatible beforehand.

Another algorithm is called backjumping [7]. It differs from backtracking in the following: when it is necessary to return to the previous step, the algorithm performs analysis and returns immediately to the variable assigning to which led to the absence of consistent values at the current step.

When assigning each variable, the algorithm called forward checking [3] removes those values that are incompatible with the values of already assigned variables from the domains of non-assigned variables. Here, if the domain of any variable becomes empty, the latest assigned value is considered incompatible.

There also exist algorithms that call a narrowing algorithm after the assigning of each variable. These algorithms, along with the forward checking, are called lookahead-algorithms [11].

Another subclass of search algorithms is the class of stochastic algorithms [5, 8, 9]. They are usually used in those cases when it is necessary to find quickly an arbitrary solution of the problem under consideration. Algorithms of stochastic search are such search algorithms that include heuristics and elements of randomness. Among them, the classes of algorithms, such as the hill-climbing and connectionlist, are best known.

In the general case, the hill-climbing algorithm is determined by two functions. One of them is an evaluation function which maps each vector from the search space to a value (which is a number). The other one is an adjacency function which assigns one or several other points to each point from the search space. Maximal values of the evaluation function must be attained at solutions to the problem. The algorithm consists in the choice of a random point in the search space and the execution of a series of transitions: at each step, all the points which are adjacent according to the adjacency function are evaluated using the evaluation function. Then one of the points that have greater values than the value of the current point is chosen randomly, and the transition to that point takes place. The algorithm terminates if the values of all adjacent points are less than or equal to the value of the current point. The best known hill-climbing algorithms are the gradient-based conflict minimization [9] and the heuristic repair [5].

Most stochastic algorithms do not guarantee passage through the entire search space. This is their most serious drawback. Their major advantage is as follows: it turns out that only they can can find solutions to some problems in a reasonable time.

There exist heuristic algorithms of search, which guarantee passage through the entire search space. The algorithm of iterative broadening [1], which is a kind of backtracking, is an example of such algorithms. Its idea is to add elements of randomness to the distribution of computational efforts along the tree of search. The number of visits to each node of the tree has a certain limit. When this limit is reached, all unconsidered branches of this node are ignored. If, at a given limit of the number of visits, the algorithm does not find a solution, the limit increases.

Now let us consider the process of manual scheduling.

3. Manual scheduling

Let us consider the operation of a dispatcher making a schedule for a university. The process of scheduling consists, as in the case of backtracking, in a successive assignment of the time, the necessary resources, such as classrooms, and, possibly, the lecturers to each lesson. In this case, lectures which require many resources (several groups) are scheduled first. Lessons in subgroups are scheduled last. In scheduling the lessons, each new lesson is put to a free place if possible, that is, it is placed so as not to be in conflict with the lessons already scheduled in terms of resources.

If there is no free place, some previously scheduled lessons are to be removed from the schedule. In this case, the simplest lessons for which it will be the easiest to find a new time or assign a new resource (for instance, another classroom) are removed. Lectures and lessons the scheduling of which took a good deal of effort of the dispatcher are not removed usually.

Thus, some characteristics determining the scheduling complexity can be assigned to each lesson. Let us introduce a concept of the cost of a lesson. We distinguish between the initial cost of a lesson and its current cost. The initial cost of a lesson reflects the preliminary estimated complexity of scheduling the lesson. It should be calculated on the basis of the specific and number of constraints imposed on this lesson. It should not change in the process of scheduling. It is clear that the more resources are used by this lesson, the more constraints are imposed on it and the more difficult is to find the proper time for it. Thus, the initial cost of lectures will be higher than the initial cost of lessons in subgroups.

For the lessons that are to be put into the schedule, we can introduce a concept of a current cost. The current cost of a lesson must reflect how easy it was to find a place for the lesson. That is, if, for instance, a lecture had to be removed from the schedule in order to arrange a lesson, it can be concluded that the scheduling was not easy, and that it is better not to reschedule this lesson further. Thus, the current cost depends on the initial cost and on the current costs of all the lessons that had to be removed from the schedule in order to insert this lesson.

And finally, before choosing a position for a given lesson, the dispatcher estimates all possible variants and chooses the simplest one. We can introduce a concept of the cost of a position. In the scheduling of a given lesson, the position cost is equal to zero if we do not violate any constraint that connects the lesson with the other scheduled lessons. Otherwise, the position cost depends on the number and current costs of those lessons that are to be removed from the schedule in order to put the lesson to this position.

Thus, manual scheduling can be approximately described as follows. The lesson with maximal initial cost is chosen from the set of non-scheduled lessons. If there are several lessons with maximal initial cost, any of them is chosen. From all possible positions, the position with minimal cost is chosen for a given lesson, and the lesson is put to this position. All previously scheduled lessons that are in conflict with the newly scheduled one on some resource (for instance, classroom, group, lecturer) are removed from the schedule. We memorize how difficult it was to schedule the lesson. This is done in order to decide whether it should be removed later. After this, we choose another non-scheduled lesson, and proceed until all lessons are scheduled.

As for position costs and initial costs of lessons, it can be said that they are determined intuitively by the dispatcher from his experience of work. In the next section, we present an NRS-algorithm based on the method of dispatchers operation described above.

4. Algorithm of non-return search

Definition 1. A constraint satisfaction problem (CSP) is a tripple $M = (X, D, C)$, where

X is a set of variables $\{x_1, \ldots, x_n\}$,

$D = D_1 \times \ldots \times D_n$, D_i is the domain of the variable x_i,

C is a set of constraints $\{c_1, \ldots, c_m\}$ on all or some of the variables $x_1, \ldots, x_n$.

In what follows, we consider only those problems that have finite sets of possible values of variables and in which all constraints are binary.

Definition 2. A vector $(a_1, \ldots, a_n) \in D$ is a solution to a given CSP if for all constraints $c_l \in C$ the condition $(a_{l_1}, \ldots, a_{l_u}) \in c_l$ holds, where $l_1, \ldots, l_u$ are the subscripts of the variables that are bound by the constraint c_l.

The algorithm of non-return search, like backtracking, is a successive assigning of certain values to the variables. If a value assigned to a variable is incompatible with the values of the previously assigned variables, these variables are set unassigned. To determine the rules for choosing a variable and a value for assigning, we introduce the notions of initial and current costs of a variable, as well as the notion of the cost of its value. The following three constants are used to define the notions: $P_{initial}, P_{max}$ and $P_{unassign}$. The constants should satisfy the following conditions: $P_{unassign} \geq 0$, $0 < P_{initial} < P_{max}$. Later we return to these constants.

Thus, let the set of constraints of the problem contain k constraints binding a variable x_i. Then the number $kP_{initial}$ will be called the *initial cost* of the variable x_i.

The notion of the current cost of a variable is determined recursively. If a variable x_i has not been ever assigned, its current cost is equal to the initial one. Otherwise, at each assigning its current cost increases as described below.

Let us assume that the set of assigned variables, before assigning to a variable x_i, consisted of variables $x_1, \ldots, x_{i-1}$. Let a_i be the value assigned to the variable x_i, p_i be the initial cost of the variable x_i, and P_i be the current cost before assigning. Let $x_{j_1}, \ldots, x_{j_k}$ be those assigned variables the values of which are in conflict with the value a_i of the variable x_i, and let $P_{j_1}, \ldots, P_{j_k}$ be their current costs.

Then the number

$$p_i + P_i + \sum_{m=1}^{k} P_{j_m}$$

is the *current cost* of the variable x_i after assigning.

Finally, let us define a notion of the cost of a value. Let us assume that the set of assigned variables before assigning to a variable x_i consisted of the variables $x_1, \ldots, x_{i-1}$. Let a_i be a value from the domain of possible values of the variable x_i. Let $x_{j_1}, \ldots, x_{j_k}$ be the variables the values of which are in conflict with the value a_i of the variable x_i. Let $P_{j_1}, \ldots, P_{j_k}$ be their current costs.

Then the number

$$kP_{unassign} + \sum_{m=1}^{k} P_{j_m}$$

is called the *cost of the value* a_i.

Now, we can formulate the algorithm of non-return search. Thus, initially all variables are said to be unassigned. The algorithm consists of successive iterations. The following actions are performed at each iteration:

- search for the maximal current cost of unassigned variables;
- compilation of a list of the unassigned variables whose current costs are equal to the maximal one;
- random choice of some variable from the list; the quality of the generator of pseudorandom numbers used is of importance here;
- search for a value with the least cost of the chosen variable; if there are several such values, we choose any of them;
- assigning of the found value to the chosen variable;
- unassigning variables whose values conflict with the newly assigned value of the chosen variable.

The algorithm terminates its work if all variables are assigned, or if the cost of the value to be assigned exceeds P_{max}. In the first case, a solution is found, and in the second case no solution is found.

Let us estimate the number of iterations that can be performed by the NRS-algorithm. Let n be the number of variables in the problem. Since only one variable is assigned at each iteration, the total number of iterations of the algorithm cannot be less than n.

Let us assume that the algorithm does not terminate its work in n iterations. Let us consider a series of $n+1$ successive iterations. Let p_i, where $1 \leq i \leq n+1$, be the current cost before the i-th iteration of the variable assigned at the i-th iteration of this series. Since the total number of variables of the problem is n, at least one of the variables was assigned twice during these $n+1$ iterations. Let the same variable be assigned at the iterations i and j of this series, where $i < j$. Then, according to the definition of the current cost and the inequality $P_{initial} > 0$, we have $p_i < p_j$. Hence, there exists an integer k such that $1 \leq k \leq n$ and $p_k < p_{k+1}$. Let x be the variable assigned at the iteration k and y be the variable assigned at the iteration $k+1$.

Since an unassigned variable with maximal current cost is chosen, the variable y was in the assigned state before the iteration k. Hence, it was unassigned at the iteration k, and consequently the current cost of the variable x at the iteration k is at least doubled. Notice also that $p_{k+1} < P_{max}$. Otherwise, the cost of the value assigned to the variable x would exceed P_{max}, and in this case the algorithm would terminate its work.

Thus, we conclude that at one of n iterations the current cost of at least one variable is at least doubled. And before increasing, this cost did not exceed P_{max}. Hence, it can be concluded that the total number of iterations performed by the algorithm does not exceed $n^2(1+\log_2 P_{max} - \log_2 P_{initial})$.

The numbers 1, 10^{300}, and 10^4, respectively, were used for $P_{initial}, P_{max}$ and $P_{unassign}$. These values were chosen experimentally. The results of testing of the algorithm with these constants are presented in the next section.

5. Results of experiments

In this section we compare the execution time for solving N-queens and map coloring problems using a software system called S3 Solver with a built-in algorithm of non-return search and using an ILOG Solver.

The statement of a map coloring problem was taken from the site [12]. In this statement, a $(N+1) \times N$ matrix with a given odd N is constructed. Its elements are determined by the following formula:

$$a_{i,j} = \begin{cases} (i-1)N - \frac{(i-1)i}{2} + j, & \text{if } i \geq j, \\ (j-1)N - \frac{(j-1)j}{2} + i - 1, & \text{otherwise.} \end{cases}$$

The problem has $N(N+1)/2$ variables: $x_1, \ldots, x_{N(N+1)/2}$. It is formulated as follows:

$$\forall i \in \{1, \ldots, N+1\}\ \forall j, k \in \{1, \ldots, N\}\ j \neq k \Rightarrow x_{a_{i,j}} \neq x_{a_{i,k}}.$$

The time for solving the problems by the NRS-algorithm is given in the table below. It was obtained at a workstation Ultra-60. The time for solving the problems with the help of ILOG was taken from the site [12] and is also presented in this table.

problem	NRS, s	ILOG, s
Queens-100	0.02	0.12
Queens-500	0.09	3.73
Queens-1000	0.21	18.94
Map-19	0.01	0.04
Map-25	0.02	0.11
Map-41	0.03	94.21

This table shows that the NRS-algorithm is highly efficient. In spite of the fact that the NRS-algorithm does not guarantee a solution in the general case, the experiments performed have shown that the problem of N-queens is successfully solved at $4 \leq N \leq 100\,000$ and at all N that are multiples of 1000 such that $100\,000 \leq N \leq 2\,000\,000$. The problem of graph coloring is solved successfully for all odd N such that $3 \leq N \leq 1499$.

The compilation of the N-queens problem from a textual representation into an internal representation of the NRS-algorithm takes most of the time in the solution of the problem. Therefore, experiments were performed under the condition that the constraints of the problem had been built into the NRS-algorithm beforehand. The results obtained are presented in the table below.

N	NRS, s
100'000	0.07
500'000	0.81
1'000'000	1.78
2'000'000	3.82
3'000'000	8.86

Thus, if the constraints are built into the NRS-algorithm, we can solve the problem of queens in approximately the same time as with the help of the well-known algorithm QS4 (Queen Search 4) [10]. This algorithm is specially designed to solve this problem. The algorithm QS4 needs 38 seconds to solve a one-million-queens problem on Sparc 1 and 17 seconds on IBM RS 6000. Sparc 1 is approximately 32 times slower than Ultra-60. That is, the time for solving a one-million-queens problem with the help of the NRS-algorithm on Sparc 1 is approximately 57 seconds.

As for other methods, the heuristic repair gives the best time for solving the problem of queens. It solves it in 90–240 seconds on Sparc 1 [5]. These results were also obtained in implementation of a special optimization of the algorithm allowing for the specific character of the problem being solved. The algorithm itself, however, can also be used to solve other problems.

Thus, the experiments have shown that, in spite of its simplicity, the NRS-algorithm is superior to the algorithms presently known on the tests carried out in this study. To check experimentally the number of iterations performed by the NRS-algorithm when it does not find a solution, the algorithm was used to place N queens on an $N \times (N-1)$ chessboard. The results of the experiments are presented in the table below.

N	No. of iterations (i)	i/N
4	392	98.00
8	1062	132.75
16	2439	152.43
32	5098	159.31
64	10399	162.48
128	21056	164.50
256	42350	165.43
512	84847	165.72
1024	169837	165.86
2048	339851	165.94
4096	679807	165.97

It is seen from the table that the dependence of the number of iterations on the number of variables is close to linear.

6. Conclusion

An algorithm of non-return search (the NRS-algorithm), a stochastic search algorithm, is presented. It is designed to solve finite binary CSPs where constraints of the "not equal" type predominate.

The experiments performed have shown that, in comparison with the algorithms based on backtracking, the NRS-algorithm is superior when it is used to solve the problems of large dimensions with many solutions. In a relatively short time, the algorithm either finds a solution or terminates its work. Although the NRS-algorithm, as most stochastic algorithms, does not guarantee a solution, the results of experiments presented above have shown that it still finds a solution for many problems.

In spite of the fact that the algorithm was formulated for binary CSPs, it can be generalized for arbitrary constraints. This work is being carried out by the author. The NRS-algorithm is built into a scheduling system, which is now under development.

References

[1] Ginsberg M. L., Frank M., Halpin M. P., Torrance M. C. Search lessons learned from crossword puzzles // Proc. of the Eight National Conf. on Artificial Intelligence. — San Francisco: Freeman, 1990. — P. 210–215.

[2] Golomb S. W. , Baumert L. D. Backtrack programming // J. of the ACM. — 1965. — Vol 12, N. 4. — P. 516–524.

[3] Haralick, R. M. Elliot G. L. Increasing the search efficiency for constraint satisfaction problems // AI. — 1980. — Vol. 14, N. 3. — P. 263–313.

[4] Kumar V. Algorithms for constraint-satisfaction problems: a survey // AI. — 1992. — Vol. 13, N. 1. — P. 32–44.

[5] Minton S., Johnston M. D., Philips A. B., Laird P. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. // Proc. of AAAI90. — 1990. — P. 17–24.

[6] Nilsson N. J. Principles of artificial intelligence. — Palo Alto: Tioga, 1980.

[7] Prosser P. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping: Research Rep. 95/177, Department of Computer Science. — University of Strathclyde, 1995.

[8] Selman B., Levesque H., Mitchell D. A new method for solving hard satisfiability problems // Proc. of the 10th National Conf. on AI, San Jose, USA. — 1992. — P. 440–446.

[9] Smith B. M. Filling the gaps: reassignment heuristics for constraint satisfaction problems: Rep. N. 92.29, School of Computer Studies. — University of Leeds, November 1992.

[10] Sosic R., Gu J. 3,000,000 queens in less than one minute // SIGART Bulletin. — 1991. — Vol. 2. N. 2, P. 22–24.

[11] Tsang E. Foundations of constraint satisfaction. — London: Academic Press, 1993. — 421 p.

[12] http://www.intelengine.com