A Genetic Local Search Algorithm for Random Binary Constraint Satisfaction Problems

Elena Marchiori
Faculty of Sciences
Free University Amsterdam
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
elena@cs.vu.nl

Adri Steenbeek
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
adri@cwi.nl

ABSTRACT

This paper introduces a genetic local search algorithm for binary constraint satisfaction problems. The core of the algorithm consists of an ad-hoc optimization procedure followed by the application of blind genetic operators. A standard set of benchmark instances is used in order to assess the performance of the algorithm. The results indicate that this apparently naive hybridation of a genetic algorithm with local search yields a rather powerful heuristic algorithm for random binary constraint satisfaction problems.

Categories and Subject Descriptors

G.1.6 [**Mathematics of Computing**]: Optimization—*Global Optimization*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms, Experimentation

1. INTRODUCTION

In the binary constraint satisfaction problem (BCSP) we are given a set of variables, where each variable has a domain of values, and a set of constraints acting between pairs of variables. The problem consists of finding an assignment of values to variables in such a way that the restrictions imposed by the constraints are satisfied. In this paper we consider random binary constraint satisfaction problems (RBCSP), since their properties in terms of difficulty to be solved have been well-understood and hence such constraints have been used for testing the performance of algorithms for solving BCSPs. More specifically, a class of random binary CSPs can be described by means of four parameters $\langle n, m, d, t\rangle$, where n is the number of variables, m is the (uniform) domain size, d is the probability that a constraint exists between two variables, and t is the probability of a conflict between two values across a constraint. CSPs exhibit a *phase transition* when a parameter is varied. At the phase transition, problems change from being relatively easy to solve (i.e., almost all problems have many solutions) to being very easy to prove unsolvable (i.e., almost all problems have no solutions). The term *mushy region* is used to indicate that region where the probability that a problem is soluble changes from almost zero to almost one. Within the mushy region, problems are in general difficult to solve or to prove unsolvable. Recent theoretical investigations ([22; 23]) allow one to predict where the hardest problem instances should occur. These predictions have been empirically supported for higher density/tightness of the constraint networks ([18]).

An heuristic algorithm considers a BCSP as a combinatorial optimization problem: the objective is to find an instantiation of values for the variables which maximizes the number of constraints that are satisfied. However, the search for such an assignment does not guarantee to converge to a global optimum. As a consequence, heuristic algorithms cannot in general detect unsatisfiability.

A rather popular class of heuristic algorithms consists of so-called genetic algorithms. A genetic algorithm (GA) is a population based iterative stochastic technique for solving combinatorial optimization problems ([9; 14]). In recent years, a number of techniques for solving CSPs based on genetic algorithms have been proposed (e.g., [2; 5; 6; 7; 19; 20]). The majority of these algorithms incorporate heuristic information into the fitness function and/or into the GA operators (selection, crossover, and mutation). For instance, in [19; 20] a fitness function that uses information about the connectivity of the constraint network is employed.

The aim of this paper is to show how one can obtain a simpler yet more effective GA based heuristic algorithm for BCSPs by using a technique where heuristic information is not incorporated into the GA operators, but is included into the GA as a separate module. The idea of incorporating heuristics into genetic algorithms for solving combinatorial optimization problems is not new [11; 17], and it has been successfully applied to various different combinatorial optimization problems. The approach we employ is known as genetic local search (GLS) (or more in general, memetic search) [13; 16]. We design a novel GLS algorithm which consists of the repeated application of the following two steps to a popula-

SAC 2000 Como, Italy

tion of candidate solutions. First, a local optimization procedure is applied to each candidate solution. Next blind GA operators (selection, crossover, mutation and replacement) are applied to the resulting population.
Extensive experiments conducted on randomly generated BCSPs support the effectiveness of this approach for solving random binary constraint satisfaction problems.
The rest of the paper is organized as follows. The next section introduces the genetic local search algorithm for BCSPs. Section 3 contains an experimental comparative analysis of the algorithm. Finally, Section 4 summarizes the results of the paper.

2. GENETIC LOCAL SEARCH FOR BCSPS

Genetic local search (GLS) is a population based iterative search scheme for combinatorial optimization problems.
Roughly, it consists of the application of genetic operators to a population of local optima produced by a local search procedure. The process is iterated until either a solution is generated or a maximal number of generations is reached. Genetic local search has been applied with success to various paradigmatic combinatorial optimization problems (e.g., [12; 13]).

```
BEGIN
 t := 0;
 initialize P(t);
 (*)apply local search to P(t);
 evaluate P(t);
 WHILE (NOT termination-condition) DO
  BEGIN
   t := t+1;
   WHILE (|P(t)| < |P(t-1)|) DO
    BEGIN
     select parents from P(t-1);
     recombine parents
     mutate children
     (*)apply local search to children
     insert children into P(t)
    END
  END
END
```

The GLS scheme that is used in our algorithm is illustrated above, where $|P(t)|$ denotes the number of elements of the set $P(t)$.
In order to design our GLS algorithm for binary CSPs we can specify the local search algorithm and the genetic algorithm features separately. The resulting algorithm is called `RIGA` (Repair-Improve Genetic Algorithm).

2.1 Notation and Terminology

A *binary CSP* is a triple $(V, \mathcal{D}, C)$ where $V = \{x_1, \ldots, x_n\}$ is a set of variables, $\mathcal{D} = (D_1, \ldots, D_n)$ is a sequence of finite domains, such that x_i takes value from D_i, and C is a set of binary constraints. A *binary constraint* c_{ij} is a subset of the Cartesian product $D_i \times D_j$ consisting of the compatible pairs of values for (x_i, x_j).
For simplicity here and in the sequel we shall assume that all the domains D_i are equal ($D_i = D$ for $i \in [1, n]$).
An *instantiation* α of a set of variables $S = \{x_1, \ldots, x_k\}$ is a mapping $\alpha : S \to D$, where $\alpha(x_i)$ is the value associated to x_i. (the notation $\alpha = \{x_1/v_1, \ldots, x_n/v_n\}$ will also be used, meaning $\alpha(x_i) = v_i$ for $i \in [1, n]$).
We call α *partial instantiation* if $\alpha(x_i)$ is not defined for some x_i in S. In such a case x_i is said to be *uninstantiated*.
A *partial solution σ of a CSP with respect to a set $S \subseteq V$* is an instantiation of S such that $(\sigma(x_i), \sigma(x_j))$ is in c_{ij}, for every x_i, x_j in S with $i \neq j$. A *solution of a CSP* is a partial solution with respect to V.
A partial solution σ with respect to S is called *maximal* if for every $x \notin S$ we have that $\sigma \cup \{x/v\}$ is not a partial solution, for every $v \in D$.
Given a partial solution σ of a CSP with respect to S, we say that a variable x_i in S has *conflict* v with a variable x_j if v is in the domain of x_j and $(\sigma(x_i), v)$ is not in c_{ij}. The set of conflicts of x_i with x_j with respect to σ is defined by $conf_\sigma(x_i, x_j) = \{v \in D \mid x_i \text{ has conflict } v \text{ with } x_j \text{ wrt } \sigma\}$. Moreover, we define the *conflict number of x with S with respect to σ* by $nconf_\sigma(x, S) = \sum_{y \in S} |conf_\sigma(x, y)|$.

2.2 The Repair-Improve Heuristic

In order to solve a CSP, we use a problem representation as in [1; 21], which associates a subset X_i of the domain D to each variable x_i, called *actual domain of x_i*. Observe that X_i can be empty, meaning that the corresponding variable x_i is uninstantiated. Using this representation, a candidate solution X is a sequence $(X_1, \ldots, X_n)$, with $X_i \subseteq D$, describing the set of partial instantiations σ such that for $i \in [1, n]$ if $\sigma(x_i)$ is defined then $\sigma(x_i) \in X_i$.
The local search algorithm used in `RIGA` takes as input a candidate solution X and transforms it into a maximal partial solution using the following algorithm. The algorithm uses as variables the candidate solution X, a partial instantiation σ, and sets of variables S, S'. The symbol $\leftarrow$ is used below to denote the assignment statement.

Input. A candidate solution $X = (X_1, \ldots, X_n)$.

Output. A maximal partial solution with respect to S' represented by (the transformed) X.

Method. Consists of the following three phases:

1. **Initialization.** $\sigma \leftarrow \emptyset$; $S \leftarrow \{x_i \in V \mid X_i \neq \emptyset\}$; $S' \leftarrow \emptyset$;
2. **Repair.** Consists of the following two steps:
   (a) **extract.** For each $x_i \in S$:
   stop $\leftarrow$ *false*;
   repeat the following three steps until *stop* = *true* or $X_i = \emptyset$:
   - select randomly $v \in X_i$;
   - $X_i \leftarrow X_i \setminus \{v\}$;
   - if $\sigma \cup \{x_i/v\}$ is a partial solution wrt $S' \cup \{x_i\}$ then:
   $\sigma(x_i) \leftarrow v$, $S' \leftarrow S' \cup \{x_i\}$, $X_i \leftarrow \{v\}$, and *stop* $\leftarrow$ *true*.
   (b) **extend.** For each $x_i \in V \setminus S'$:
   $T \leftarrow D$;
   stop $\leftarrow$ *false*;
   repeat the following three steps until *stop* = *true* or $T = \emptyset$:
   - select randomly $v \in T$;
   - $T \leftarrow T \setminus \{v\}$;
   - if $\sigma \cup \{x_i/v\}$ is a partial solution wrt $S' \cup \{x_i\}$ then:

$\sigma(x_i) \leftarrow v$, $S' \leftarrow S' \cup \{x_i\}$, $X_i \leftarrow \{v\}$, and $stop \leftarrow true$.

3. **Improve.** Consists of the following three steps:

   (a) **arc-consistency.** For each x in S':
   if $conf_\sigma(x, y) = D$ for at least one variable $y \neq x$ then
   $\sigma \leftarrow \sigma \setminus \{x/\sigma(x)\}$, $X_i \leftarrow \emptyset$, and $S' \leftarrow S' \setminus \{x\}$.

   (b) **delete.**[1] Let
   $maxc_\sigma(S') = max_{x \in S'}\ nconf_\sigma(x, V \setminus S')$,
   $T = \{x \in V \mid nconf_\sigma(x, V \setminus S') = maxc_\sigma(S')\}$.
   Select randomly one variable x in T:
   $\sigma \leftarrow \sigma \setminus \{x/\sigma(x)\}$, $X_i \leftarrow \emptyset$, and $S' \leftarrow S' \setminus \{x\}$.

   (c) **extend.** (As extend step above defined).

One can check that the computational complexity of the Repair-Improve Heuristic is $O(n \cdot |D| \cdot n\_constraints)$ where *n_constraints* denotes the number of binary constraints in the CSP problem.
It is not difficult to show that after application of `Repair` X represents a maximal partial solution with respect to S'.
Then `Improve` tries to find a better X by searching in the 0: first, it removes those assignments of σ that cannot be part of any solution (arc-consistency step): next, it removes the value of one variable having the most conflicts (delete step). Finally, it applies the extend step to the resulting partial solution.
After application of `Improve` X represents a maximal partial solution with respect to S'.
Observe that `arc-consistency` and `delete` are the only steps of `RIGA` that use information on the conflicts among variables in the CSP.

2.3 The Genetic Algorithm

The main features of the genetic algorithm component of `RIGA` can be summarized as follows:

Representation. A chromosome is a candidate solution.

Fitness. The fitness of a chromosome is equal to the number of instantiated variables in the chromosome.

GA type. Generational genetic algorithm (see the pseudocode at the beginning of the section) with elitist selection mechanism which copies the best individual of a population to the population of the next generation [10].

Genetic operators. We use the following two GA reproduction operators:

crossover. (always applied): two offsprings O and O' are incrementally constructed from two parents P and P', where the i-th components O_i, O'_i of O, O' are incrementally constructed from P_i, P'_i starting from $O_i = O'_i = \emptyset$ by adding each element $v \in P_i \cup P'_i$ either to O_i or to O'_i with equal probability.

mutation: two mutation operators are applied to a chromosome X. Mutation 1 (applied to each X_i with probability $1/n$, where n denotes the number of variables): add a randomly chosen value of D to X_i. Mutation 2 (applied to each X_i with low probability (typical value 0.05)): remove a randomly chosen element from X_i.

It is worth to note that the genetic operators are blind, that is they do not use information about the CSP, and they perform all choices in a random way. This is counterbalanced by the Repair-Improve heuristic, which transforms chromosomes into maximal partial solutions.
We use a rather small population, consisting of 10 individuals. This choice is justified by the results obtained in our experiments with different population sizes.

3. EXPERIMENTAL RESULTS

In order to assess the effectiveness of `RIGA`, we compare experimentally `RIGA` and the heuristic based GA algorithm introduced in [4], here called `MIDA`, which to the best of our knowledge is the best genetic based heuristic algorithm for BCSPs.
`MIDA` incorporates heuristics in the reproduction mechanism and in the fitness function in order to direct the search towards better individuals. More precisely, `MIDA` works on a pool of 8 individuals. As `RIGA`, it uses a roulette-wheel based selection mechanism; however, it is not generational, but has a steady state reproduction mechanism where at each generation an offspring is created by mutating a specific gene of the selected chromosome, called pivot gene, and that offspring replaces the worse individual of the actual population. Roughly, the fitness function of a chromosome is determined by adding a suitable penalty term to the number of constraint violations the chromosome is involved in. The penalty term depends on the set of breakouts whose values occur in the chromosome. A breakout consists of two parts: 1) a pair of values that violates a constraint; 2) a weight associated to that pair. The set of breakouts is initially empty and it is modified during the execution by increasing the weights of breakouts and by adding new breakouts according to the technique used in the Iterative Descent Method ([15]). Therefore we have named this algorithm `MIDA`, standing for Microgenetic Iterative Descent Algorithm.
In [4] it is shown that `MIDA` outperforms the Iterative Descent Method algorithm [15]. Moreover, according to the recent works [3; 8] on the experimental comparison of GA based algorithms for CSPs, `MIDA` results to be the best GA based algorithm for RBCSPs.
We perform extensive experiments on random binary CSPs with 15 variables and uniform domains of 15 elements. These values are common in the empirical study of GA based algorithms for (random) BCSPs ([5]). Later on we will discuss the effect of varying the number of variables.
We use the generator by van Hemert (cf. [8]) for constructing random BCSP instances. The generator of these instances first calculates the number of constraints that will be produced using the equation $\frac{n(n-1)}{2} \cdot d$. It then starts producing constraints by randomly choosing two variables and assigning a constraint between them. When a constraint is assigned between variable x_i and x_j, a table of conflicting values is generated. To produce a conflict two values are chosen randomly, one for the first and one for the second variable. When no conflict is present between the two values for the variables, a conflict is produced. The number

[1] This step is applied with high probability (typical value 0.9).

of conflicts in this table is determined in advance by the equation $m \cdot m \cdot t$.

RIGA and MID are tested on the same 925 problem instances: 625 instances used in [8] (see Table 1), obtained by considering 25 instances for each combination of density d and tightness t with $d, t \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$; and 300 instances close to the mushy region, obtained by considering 25 instances for each of the 12 combinations reported in Table 2. We executed 10 independent runs on each instance.

The algorithms performance is evaluated by means of two measures. The *Success Rate* (SR) is the fraction of instances where a solution has been found. The *Average number of Evaluations to Solution* (AES) is the number of fitness evaluations, i.e. the number of newly generated candidate solutions in *successful* runs. The algorithms terminate if a solution is found or the limit of 100000 generated candidate solutions is reached.

d	t	MIDA	RIGA
0.2	0.8	0.52 (15304)	0.54 (6833)
0.2	0.9	0.0 (-)	0.0 (-)
0.3	0.6	1 (2625)	1 (413)
0.4	0.5	1 (2573)	1 (272)
0.4	0.6	0.29 (43656)	0.46 (21532)
0.5	0.4	1 (958)	1 (74)
0.6	0.4	1 (6029)	1 (651)
0.6	0.5	0.0 (-)	0.0 (-)
0.7	0.4	0.80 (30775)	1 (5642)
0.8	0.3	1 (1999)	1 (151)
0.8	0.4	0.11 (51149)	0.16 (14427)
0.9	0.4	0.0 (-)	0.0 (-)

Table 2: Results for MIDA and RIGA

Tables 1 and 2 summarize the results of the experiments, where the value of AES is given between brackets. Considering the results from the point of view of the problem instances, we can observe a distribution of the success rates which is in accordance with the theoretical predictions of the phase transition for binary CSP problems reported in [22; 23].

Considering the success rate, RIGA performs equally or better than MIDA in all classes of instances. Concerning the computational effort (AES), RIGA usually requires much less evaluations to find a solution than MIDA. However, the AES measure does not take into account the effort required by the heuristics. The effort required by the Repair-Improve heuristic affects the running time needed to find a solution, which can become about ten times slower than MIDA on some hard problem instances.

3.1 Discussion

Let us briefly analyze the role of the local search module for the performance of RIGA. As shown by the results of our experiments, the Repair-Improve heuristic is able alone to solve constraints belonging to 'easy' classes, like those obtained by setting $d = 0.3, t = 0.3$. However, the heuristic alone is not as effective as RIGA on 'harder' classes. For instance, the 1 version of the Repair-Improve heuristic which considers as final solution the best solution found on 2000 independent runs, yields SR equal to 0.74 on the instances of class $d = 0.1, t = 0.9$.

As one would expect, the performance of the pure GA, that is RIGA without the Repair-Improve heuristic module, is rather poor. For instance, on the problems of class $d = 0.1, t = 0.1$ the pure GA has a SR equal to 0.

It is interesting to investigate how the results scale up when we vary the number n of variables. Figure 1 illustrates how the performance of MIDA and RIGA is affected by increasing the number of variables n, when the other parameters are set to $m = 15$, $d = 0.3$, and $t = 0.3$. The x-axis represents the number of variables, and the y-axis the AES, that is the average number of evaluations to a solutions.

We consider values of n ranging from 10 till 40 with step 5 and observe that increasing the number of variables does not affect the success rates in this range of n values. The number of iterations that are needed in order to find a solution, however, is heavily affected and for both algorithms it exhibits a super-linear growth. The two curves have a similar growth rates, although up to $n = 35$ RIGA is growing at a visibly slower rate. This seems to suggest a better scale-up behavior for RIGA.

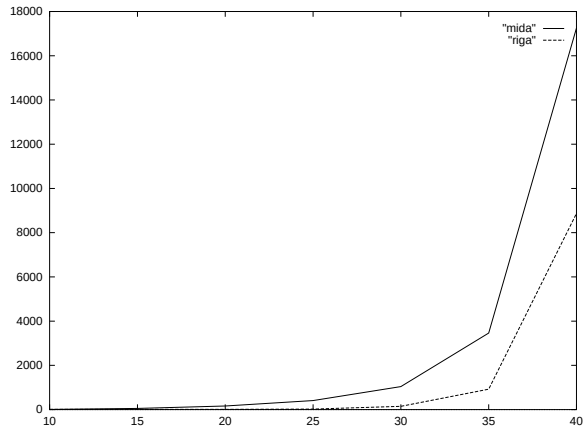

Figure 1: Scale-up values for RIGA and MIDA

4. CONCLUSION

In this paper we have introduced an effective GA based algorithm for solving BCSPs. The main novelty with respect to previous work on this subject is the use of a separate local search algorithm for improving chromosomes. Previous GA approaches for solving BCSPs have used rather sophisticated fitness functions, which bias the search towards a combination of different properties of the constraint network. In contrast, the fitness function used in RIGA describes just one property, that is the number of instantiated variables. We can use such a simple fitness function because in RIGA at each iteration the population consists of (maximal) partial solutions, thanks to the application of the Repair-Improve heuristic to the chromosomes. In this way, the search pressure determined by the fitness function is exclusively directed towards large partial solutions, while the application of the heuristic together with the genetic operators, are responsible for improving the quality of the chromosomes.

Interesting topics that remain to be investigated include the enhancement of RIGA in order to detect inconsistent BCSPs. We are aware of only one GA based algorithm that can tackle

d	alg.	t				
		0.1	0.3	0.5	0.7	0.9
0.1	MIDA	1 (1)	1 (4)	1 (21)	1 (87)	96.0 (2923)
	RIGA	1 (10)	1 (10)	1 (10)	1 (17)	96.0 (197)
0.3	MIDA	1 (3)	1 (50)	1 (323)	0.52 (32412)	0.0 (-)
	RIGA	1 (10)	1 (10)	1 (24)	0.72 (15604)	0.0 (-)
0.5	MIDA	1 (10)	1 (177)	0.90 (26792)	0.0 (-)	0.0 (-)
	RIGA	1 (10)	1 (10)	1 (6809)	0.0 (-)	0.0 (-)
0.7	MIDA	1 (20)	1 (604)	0.0 (-)	0.0 (-)	0.0 (-)
	RIGA	1 (10)	1 (42)	0.0 (-)	0.0 (-)	0.0 (-)
0.9	MIDA	1 (33)	1 (8136)	0.0 (-)	0.0 (-)	0.0 (-)
	RIGA	1 (10)	1 (588)	0.0 (-)	0.0 (-)	0.0 (-)

Table 1: Results for `MIDA` and `RIGA`

also unsatisfiable BCSP [5]. The algorithm is an extension of `MIDA` that maintains information on inconsistent values found during the execution of the GA, information that is used to detect unsatisfiability. It seems that a similar extension can be used in `RIGA`, which could perform Opportunistic Arc Revision by memorizing inconsistent values in the arc-consistency step. Moreover, it seems that the elimination of such inconsistent values would improve the running time behaviour of `RIGA` on hard instances, which contain a high number of inconsistent values.

5. REFERENCES

[1] N. Barnier and P. Brisset. Optimization by hybridation of a genetic algorithm with constraint satisfaction techniques. In *International Conference on Evolutionary Computation*, pages 645–649. IEEE, 1998.

[2] J. Bowen and G. Dozier. Solving constraint satisfaction problems using a genetic/systematic search hybrid that realizes when to quit. In Eshelman L. J., editor, *Proceedings of the 6th International Conference on Genetic Algorithms (ICGA 95)*, pages 122–129. Morgan Kaufmann, 1995.

[3] B. Craenen, A. Eiben, and E. Marchiori. Solving constraint satisfaction problems with heuristic-based evolutionary algorithms. In *Eleventh Belgium-Netherlands Conference on Artificial Intelligence*, 1999.

[4] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithms. In *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, pages 306–311. IEEE Press, 1994.

[5] G. Dozier, J. Bowen, and A. Homaifar. Solving constraint satisfaction problems using hybrid evolutionary search. *IEEE Transactions on Evolutionary Computation*, 2(1):23–33, 1998.

[6] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Constrained problems. In L. Chambers, editor, *Practical Handbook of Genetic Algorithms*, pages 307–365. CRC Press, 1995.

[7] A.E. Eiben and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pages 258–261. IEEE Press, 1996.

[8] A.E. Eiben, J.I. van Hemert, E. Marchiori, and A. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A.E. Eiben, T. Bäck, M. Schoenauer, and H. Schwefel, editors, *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature (PPSN V)*, LNCS 1498, pages 196–205. Springer, 1998.

[9] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wiley, New York, 1989.

[10] K.A. De Jong. An analysis of the behaviour of a class of genetic adaptive systems. Doctoral Dissertation, University of Michigan, Dissertation Abstract International 36(10), 5140B, 1975.

[11] A. Kolen and E. Pesch. Genetic local search in combinatorial optimization. *Discrete Applied Mathematics*, 48:273–284, 1994.

[12] E. Marchiori. A simple heuristic based genetic algorithm for the maximum clique problem. In *ACM Symposium on Applied Computing*, pages 366–373. ACM Press, 1998.

[13] P. Merz and B. Freisleben. Genetic local search for the tsp: New results. In *IEEE International Conference on Evolutionary Computation*, pages 159–164. IEEE Press, 1997.

[14] Z. Michalewicz. Genetic algorithms, numerical optimization and constraints. In L.J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms (ICGA 95)*, pages 98–108. Morgan Kaufmann, 1995.

[15] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-93*, pages 40–45. AAAI Press/The MIT Press, 1993.

[16] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical report, Caltech Concurrent Computation Program, Californian Institute of Technology, U.S.A., TR No790 1989.

[17] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7:65–85, 1988.

[18] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[19] M.C. Riff-Rojas. Evolutionary search guided by the constraint network to solve CSP. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 337–348. IEEE Press, 1997.

[20] M.C. Riff Rojas. Using the knowledge of the constraints network to design an evolutionary algorithm that solves CSP. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pages 279–284. IEEE Press, 1996.

[21] A. Ruiz-Andino and J. Ruz. Integration of constraint programming and evolution programs: Application to channel routing. pages 448–459. Springer, 1998. LNAI 1415.

[22] B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. In A.G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 100–104. John Wiley & Sons Ltd., Aug. 1994.

[23] C.P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.