Measuring the Searched Space to Guide Efficiency: The Principle and Evidence on Constraint Satisfaction

Jano I. van Hemert[1] and Thomas Bäck[1,2]

[1] Leiden Institute of Advanced Computer Science — Leiden University, Niels Bohrweg 1, 2333 CA, Leiden {jvhemert,baeck}@cs.leidenuniv.nl
[2] Nutech Solutions GmbH, Martin-Schmeisser-Weg 15, D-44227 Dortmund, baeck@nutechsolutions.de

Abstract. In this paper we present a new tool to measure the efficiency of evolutionary algorithms by storing the whole searched space of a run, a process whereby we gain insight into the number of distinct points in the state space an algorithm has visited as opposed to the number of function evaluations done within the run. This investigation demonstrates a certain inefficiency of the classical mutation operator with mutation-rate $1/l$, where l is the dimension of the state space. Furthermore we present a model for predicting this inefficiency and verify it empirically using the new tool on binary constraint satisfaction problems.

1 Introduction

Genetic operators are often taken for granted: they are used without knowledge of the statistical properties behind the process. For almost every operator it is very difficult to analytically determine its properties. In most cases doing some quick runs with a choice of favourite operators seems sufficient to produce a satisfactory result. Unfortunately this lack of knowledge may result in lower performance in the final algorithm.

As an evolutionary algorithm is developed it grows, and at the same time its complexity increases. When its development has finished, its creator starts a quest for the most efficient parameters. This optimisation problem is in principle unsolvable, but it is also necessary as many (even most) parameter settings are unreasonable or lead to an inefficient algorithm.

In the optimisation process we need to take measurements that somehow reflect the efficiency of the algorithm. Furthermore, due to stochastic differences we need to do multiple runs with each setting of parameters. Measurements on these experiments are often restricted to the accuracy, i.e., the percentage of times a solution is found within a time limit, and to the speed, i.e., the average number of evaluations it takes to find a solution. Although these properties are probably seen as most important by end users, they do not tell the developer much about what is going on inside. In general, we would like to gain insight into what is actually going wrong—especially when a system does not live up to expectations.

Here we propose a tool that provides a developer or a researcher with some insight into the process of an algorithm that is searching for a solution. This tool also aids in the verification of models of simplified evolutionary algorithms, of which we show an example.

In the next section we shall define the *searched* space (as opposed to the *search* space) of an evolutionary algorithm. Then, in Section 3, we define a mutation operator. In Section 4 we model the chance that this operator will produce unchanged copies and verify this model empirically in Section 5. Section 6 elaborates on the use of the tool in experiments on binary constraint satisfaction. We finish with conclusions in Section 7.

2 The Searched Space

When an evolutionary algorithm is performing its search, it generates candidate solutions. In essence, these are points from the whole possible state space[1]. When working with discrete domains we can define this state space in terms of the alphabet $D = \{d_1, \ldots, d_m\}$ and the number of variables l, also known as the chromosome length.

Definition 1. *State space: The total* state space *an evolutionary algorithm is able to search in is defined as* $M = D_1 \times \ldots \times D_l$. *We assume all* D_i *are equal, thus we know that* $M = D^l$ *and* $|M| = |D|^l = m^l$.

Evolutionary algorithms are stochastic optimisation tools, which means that a solution might not be found. The only proof of global convergence to a solution is made under the assumption that we have unlimited time [4]. Thus for practical reasons a limit is always set on the execution time of an evolutionary algorithm, most often in the form of a maximum number of evaluations. This maximum directly provides an upper bound to the number of points of the state space we are able to visit. In reality, however, evolutionary algorithms in discrete domains tend to visit less points than the maximum they are allowed to.

Definition 2. *Searched space of an algorithm: The* searched space $S \subset M$ *is the set of points taken from the state space that is visited by a particular evolutionary algorithm during a run.*

Insight into the searched space of an algorithm might help us in evaluating its performance. For instance, if the size of the searched space is one then the algorithm has only visited one point of the state space. Except if it found the solution in the first guess this is generally not what we want. At the other extreme, if an algorithm's searched space is just as large as the number of evaluated points in the state space, we could argue that it has fully used its potential. Nevertheless, in the case of an evolutionary algorithm, this is most probably not what we would get, as such population-based algorithms tend to lose population diversity: that is, the individuals in a population will become more alike as the number of evaluations increases. To measure this we introduce *resampling ratio*:

[1] We use the term *state space* as opposed to *search space* to prevent confusion.

Definition 3. *Resampling ratio: First we define a* revisit *as a point in the state space that we have seen before, i.e., it is already present in the searched space. The* resampling ratio *is defined as the total number of revisits in a run divided by the total number of evaluations in the same run: resampling ratio = revisits/evaluations.*

Of course due to its stochastic nature, an evolutionary algorithm has many searched spaces. With a probability extremely close to one, for every run with a different setting of the random number generator, another searched space is produced. It is not the precise content of the searched space we examine here, but its size related to the total number of evaluations performed.

3 Mutation k/l

A well known way of mutating chromosomes over a discrete alphabet is by changing every gene of the chromosome with some fixed probability $p_{mutation} = k/l$. This probability is often fixed by $k = 1$ because of both theoretical and empirical evidence that show this is a near optimal setting for the mutation rate for many problems with $m = 2$ [2, 7].

The mutation operator is shown in Algorithm 1. For every gene in the chromosome a dice is thrown with l sides. If it shows a value equal or lower than k we generate a new value for the gene by drawing a value uniform randomly from the alphabet minus the current value. This way we make sure that a different value will be chosen.

Algorithm 1 Mutation operator with probability k/l

```
for gene = 1 to l do
  if (uniform_random(l) <= k) then
    c[gene] = uniform_random(D − c[gene])
  end if
end for
```

To test and analyse the mutation operator we need to embed it in an evolutionary algorithm. Eventually we want to analyse the behaviour of the whole algorithm. As this behaviour is influenced by probabilities we will need to keep things as simple as possible. For this reason we will retain from solving a problem. Consequently no selection is performed as we have no fitness to optimise. As we select nothing we can do without a population, or in other words we use a population size of one and replace its only member in every generation with its offspring generated by the mutation operator. The whole process is written down in Algorithm 2. The lack of a selection procedure makes this algorithm not qualified to be categorised as an evolutionary algorithm. However, we would like to point out here that the resampling ratio will become only higher if a selection process is added.

Algorithm 2 Main algorithm that simulates a (1,1) strategy without selection

```
for i = 1 to l do
  c[i] = uniform_random(D_i)
end for
C = {c}
evaluations = 0
revisits = 0
while evaluations < 10,000 do
  c = mutate(c)                                   // See Algorithm 1
  evaluations = evaluations+1
  if c ∈ C then
    revisits = revisits+1
  else
    C = C ∪ {c}
  end if
end while
```

During the run we take two important measurements; the number of evaluations and the number of revisits. The set C symbolises the searched space so far created by the evolutionary algorithm, it contains every unique point that we have generated so far. Using this we can calculate the resampling ratio as revisits/evaluations or, alternatively, as $(|C| - \text{evaluations})/\text{evaluations}$.

4 A Simple Model

We show how the resampling ratio can be predicted for the algorithm in the previous section by using the conjecture that all of the revisited points are caused by the mutation operator producing unchanged individuals. That is

$$\text{resampling ratio} = P(\text{mutate}(c) = c).$$

Examining Algorithm 1 we can calculate the probability that chromosome c is not changed. Every gene $c[i]$ we look at in turn has an independent chance of being changed.

Thus we can multiply every probability of a gene not being changed to get the probability $P(\text{chromosome unchanged})$. We are left with calculating that one gene is unchanged. Whenever the mutation operator decides to change a gene it makes sure the gene gets a different value. Therefore, we need only consider the chance the gene is left unchanged.

$$P(\text{chromosome unchanged}) = \left(1 - p_{mutation}\right)^{l} = \left(\frac{l-k}{l}\right)^{l} \tag{1}$$

If we plot (1) against the length of the chromosome and setting $k = 1$ and $m = 16$ we get Figure 1. It shows that the chance of not changing converges to approximately 0.37 when l increases. We can further show this convergence by

looking at the limit when l approaches infinity:

$$\lim_{l \to +\infty} \left(\frac{l-k}{l} \right)^l = e^{-k} \tag{2}$$

For a mutation rate of $1/l$ we have $k = 1$ and that leads to:

$$P(\text{chromosome unchanged}) \approx 0.37,$$

which is very high. We propose to use a rate that allows less than 1% of unchanged chromosomes which we can achieve with $k = 5$ when l is large or a bit lower for smaller values of l.

5 Verifying the Simple Model

We want to verify our model from Section 4 by simulating the algorithm presented in Section 3. We vary the chromosome length over $2, \ldots, 16$ and let the algorithm run for 10,000 evaluations, setting $k = 1$ and $m = 16$. For every setting we do 25 independent runs with different random seeds.

If we plot the measured resampling ratio we get Figure 1 in which we see that indeed the number of revisited points in terms of ratio to the searched space approaches 0.37 when l increases. The error between the model and the measured values is quite low for a chromosome length higher than four.

The general outline of the figure corresponds with our model presented earlier except for small chromosome lengths. This behaviour can easily be explained if we examine the size of the state space in relation to the size of the searched space, as depicted in Figure 2. When the values for chromosome length become too small the size of the state space will become smaller than the number of generated points, which inevitably leads to more revisited points.

As a second test we fix m and l at 16 and vary the mutation rate $p_{mutation} = k/l$. We perform tests with $k = 1, \ldots, 5$ where for every setting of k we do 25 independent runs. Figure 2 shows the resampling ratio together with two estimations using our model (almost a perfect fit) and using the limit our model converges to.

6 Practical Evidence with CSPs

To test our hypothesis of a negative influence of the mutation rate when it is too small, we do a number of experiments on binary constraint satisfaction problems. We shall look at the behaviour of the performance and accuracy of an evolutionary algorithm when trying to solve increasingly difficult problems. We do eight experiments where we test all combinations of crossover on/off, parent selection on/off and survivor selection on/off. Each experiment will be repeated with five different mutation rates to examine the impact of k.

Constraint satisfaction problems (CSPs) form a class of models representing problems that have as common properties, a set of variables and a set of

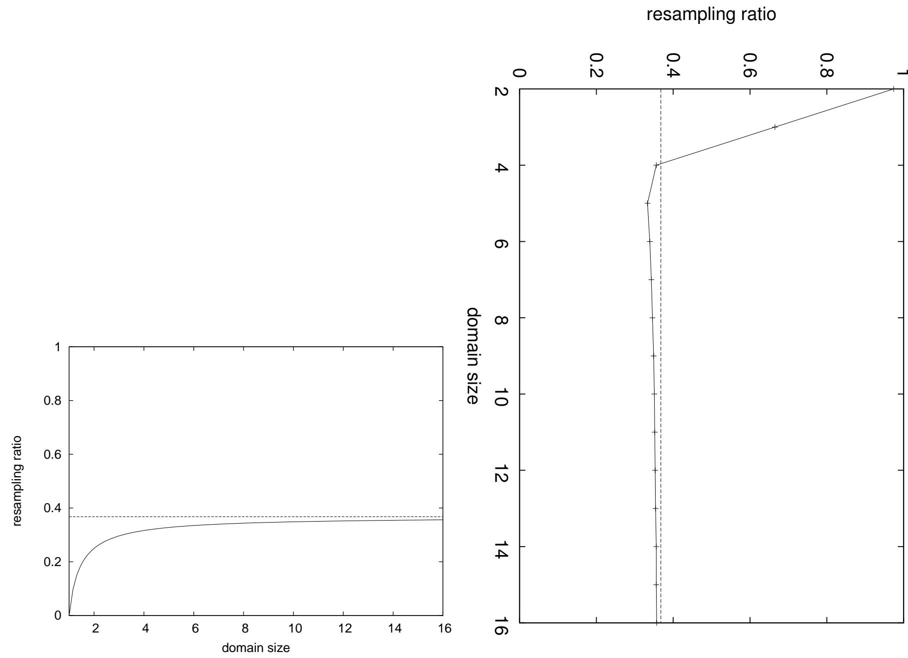


Fig. 1. Modelling the chance of not changing a chromosome (left). Simulation of the mutation operator with $p_{mutation} = 1/l$, thus $k = 1$. Every point is averaged over 25 independent runs (right).

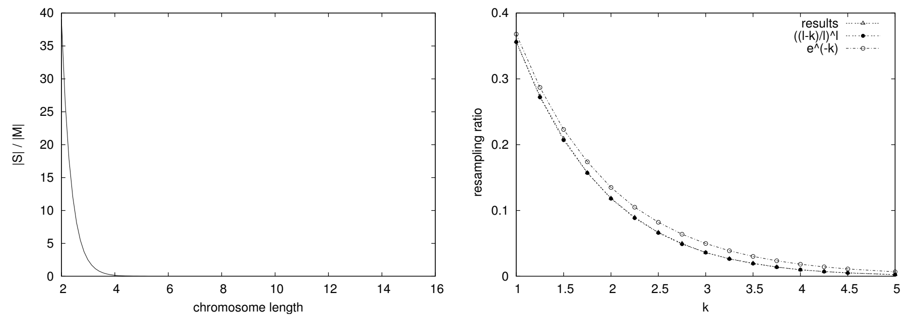


Fig. 2. Size of the searched space divided by the size of the state space (left). Varying k with $m = l = 16$ and 25 independent runs per reported result (right).

constraints. The variables should be instantiated from a discrete domain while making sure the constraints that restrict certain combinations of variable instantiations to exist, are satisfied. Examples of CSPs are k-graph colouring, 3-SAT and n-Queens.

In general, a CSP has a set of constraints where each constraint can be over any subset of the variables. Here, we focus on binary CSPs: a model that only allows constraints over a maximum of two variables. At first, this seems a restriction, but Tsang has shown [8] this is not the case proving that any CSP can be rewritten to a binary CSP. Solving the general CSP corresponds then to finding a solution for the binary form.

The last ten years many different approaches have been tried to solve constraint satisfaction problems using evolutionary algorithms. These include heuristics [3], adaptive schemes [5] and local search operators [6]. Here we fall back on the simplest approach, that is, an integer representation where each gene corresponds to one variable in the CSP. The gene can take any value from the variable's domain.

Our experiments consist of running our evolutionary algorithm with the features in Table 1 on a set of randomly created instances of binary CSPs. The table shows two selection steps, parent selection and survivor selection. The first is used to determine who is to be subjected to the genetic operators, while the second determines who is allowed to go to the next generation. These problem instances are created such that some are more difficult to solve than others. We create the test suite using the RandomCsp library [9] and choose Model E [1] as the process whereby the instances are constructed. The difficulty of the problem instances created by this process is controlled using a parameter p.

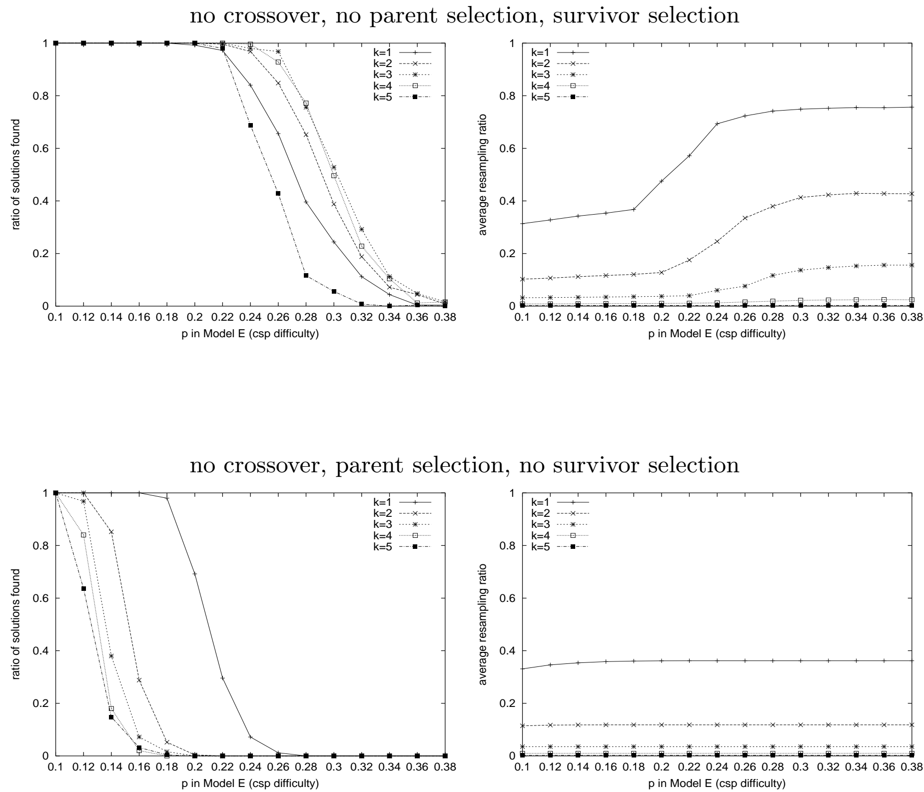
To verify the influence of different features of our evolutionary algorithm we perform eight experiments, where in each experiment we test five different mutation rates. Each test consists of many runs on many problem instances. Such a test is performed in a manner that gives insight into the behaviour of the algorithm when we use it to solve increasingly more difficult constraint satisfaction problems.

We use the same number of variables ($l = 15$) and the same domain size ($m = 15$) for each problem instance, which provides a state space with a size of 15^{15}. The parameter p determines the difficulty of an instance, which gives an overbounded estimate of the ratio of conflicts in an instance. It is varied from 0.10 to 0.38[2] in steps of 0.02, thereby increasing the difficulty of finding solutions. At every step we create 25 instances and we let the evolutionary algorithm do 10 independent runs on each instance. This totals the number of runs for each experiment to 5 mutation-rates $*$ 15 steps $*$ 25 instances $*$ 10 runs $= 18,750$ runs.

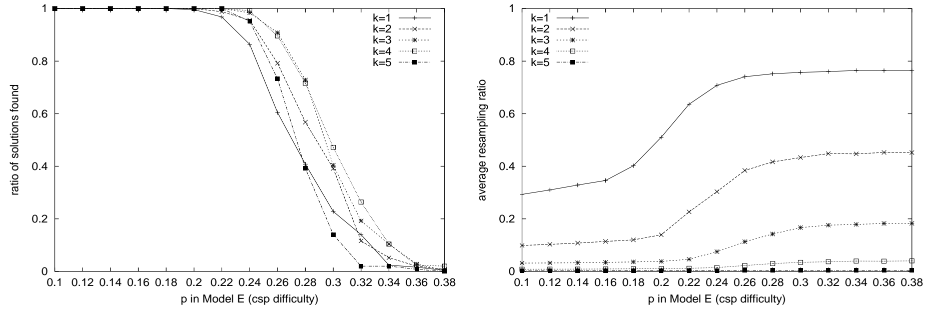
[2] The value 0.38 is close to the point beyond which we can no longer produce solvable problem instances.

Table 1. Features of the evolutionary algorithm that is used to solve binary CSPs. The features in bold font are varied in the experiments

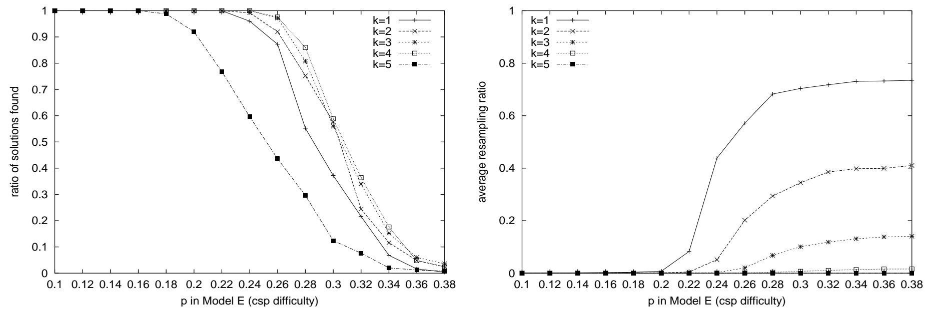
feature	*value*
representation	integer
chromosome length (l)	number of variables in CSP
fitness	number of conflicts
population size	100
evolutionary model	steady-state
parent selection	linear ranked bias (2.0) [10] *or* randomly selected
survivor selection	replace worst *or* randomly selected
mutation operator	See Algorithm 1
mutation rate	varied k in k/l
crossover operator	uniform crossover *or* none
stop criterion	solution found or 100,000 evaluations



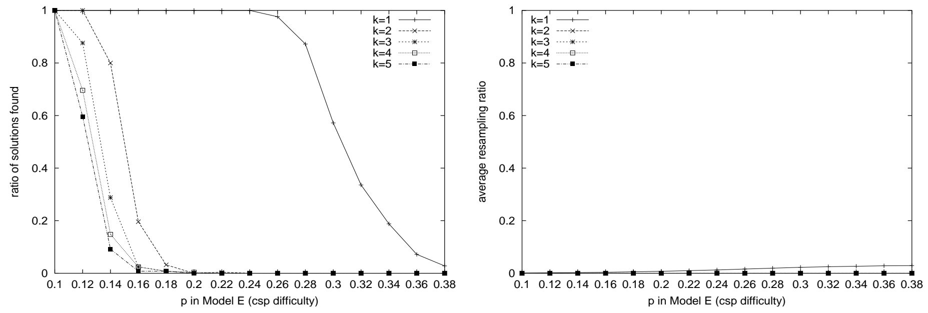
no crossover, parent selection, survivor selection



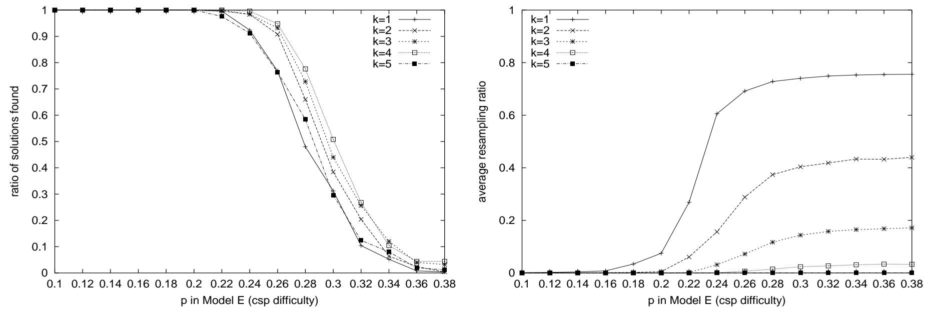
crossover, no parent selection, survivor selection



crossover, parent selection, no survivor selection



crossover, parent selection, survivor selection



During each run we measure the success rate and the resampling ratio. We present averages of these measures per experiment per setting of the mutation rate. The preceding figures are presented as follows: every row is one experiment, with two graphs, the average success rates and the average resampling ratio. Every figure contains five plots, one for each of the mutation rate settings $p_{mutation} = k/l$.

In two experiments the success rate drops to zero at $p = 0.18$, both of these have no selection methods. We leave out the results of these experiments, but we mention a difference between the resampling ratio in both experiments. With crossover the resampling ratio drops to zero for all settings of k, while without crossover we get the values predicted by the simple model.

The four experiments that have survivor selection show good performance in success rate. In all of these the best mutation rates are either $k = 3$ or $k = 4$. Using (1) and the fact that $l = 15$ we know that $P(\text{chromosome unchanged})$ is 0.0352 and 0.0095 for these two settings of k, very close to our hypothesis.

When using parent selection and not using survivor selection, the best mutation rate is $k = 1$. Furthermore, performance drops significantly when k is increased. The effect of crossover is mainly visible where we only have parent selection. There it considerably boosts performance on success rate. At the same time we witness a very low resampling ratio.

Whenever we see a good performance in success rate, we measure a low resampling ratio. However, when we look at the experiment with crossover and without any selection, we observe a low resampling ratio, but a very poor performance in success rate.

7 Conclusions

We have presented a simple measuring tool that measures independently of the evolutionary algorithm or any other algorithm that works by a repeated process of generating points in the state space. Using this tool and a theoretical model we have analysed a well known standard mutation strategy, moreover we have found that this strategy might lead to inefficient behaviour in a simplified evolutionary algorithm without selection.

To test our tool's usefulness in practical environments, we have performed a study on solving binary constraint satisfaction problems. Here the outcome is that selection and mutation rate, when wrongly balanced, can have a devastating effect on the performance. If the selection is strong and the mutation rate too weak, we observe a high resampling ratio, which means that only few points in the state space have been generated. This is a good indication of what is causing the low performance.

Although the resampling ratio might be a good way of explaining what goes wrong, it is not suitable for detecting when we may expect good performance. Thus maintaining a healthy resampling ratio is not a guarantee for a successful evolutionary algorithm in itself.

References

[1] D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction a more accurate picture. In G. Smolka, editor, *Principles and Practice of Constraint Programming — CP97*, pages 107–120. Springer-Verlag, 1997.

[2] T. Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 85–94. Springer, 1992.

[3] S.H. Clearwater and T. Hogg. Problem structure heuristics and scaling behavior for genetic algorithms. *Journal of Artificial Intelligence*, 81:327–347, 1996.

[4] A.E. Eiben, E.H.L. Aarts, , and K.M. Van Hee. Global convergence of genetic algorithms: an infinite markov chain analysis. In *Proceedings of the First International Conference on Parallel Problem Solving from Nature*, pages 4–12. Springer, Berlin, 1991.

[5] A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.

[6] E. Marchiori and A. Steenbeek. A genetic local search algorithm for random binary constraint satisfaction problems. In *ACM Symposium on Applied Computing*, pages 458–462, 2000.

[7] H. Mühlenbein. How genetic algorithms really work: Mutation and hill-climbing. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 15–25. Springer, 1992.

[8] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[9] J.I. van Hemert. *Documentation of the RandomCsp library*. Leiden University, randomcsp version 1.5 edition, 2001. Available from `http://www.liacs.nl/~jvhemert/randomcsp`.

[10] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms (ICGA'89)*, pages 116–123, San Mateo, California, 1989. Morgan Kaufmann Publishers, Inc.