How to Handle Constraints with Evolutionary Algorithms

B.G.W. Craenen **A.E. Eiben** **E. Marchiori**

Abstract

In this paper we describe evolutionary algorithms (EAs) for constraint handling. Constraint handling is not straightforward in an EA because the search operators mutation and recombination are 'blind' to constraints. Hence, there is no guarantee that if the parents satisfy some constraints the offspring will satisfy them as well. This suggests that the presence of constraints in a problem makes EAs intrinsically unsuited to solve this problem. This should especially hold when the problem does not contain an objective function to be optimized, but only constraints – the category of constraint satisfaction problems. A survey of related literature, however, indicates that there are quite a few successful attempts to evolutionary constraint satisfaction. Based on this survey we identify a number of common features in these approaches and arrive to the conclusion that EAs can be effective constraint solvers when knowledge about the constraints is incorporated either into the genetic operators, in the fitness function, or in repair mechanisms. We conclude by considering a number of key questions on research methodology.

1 Introduction

Many practical problems can be formalized as constrained (optimization) problems. These problems are in general tough (NP-hard), hence they need heuristic algorithms in order to be (approximately) solved in a short time.

EAs show a good ratio of (implementation) effort to performance, and are acknowledged as good solvers for tough problems. However, no standard EA takes constraints into account. That is, the regular search operators, mutation and recombination, in evolutionary programming, evolution strategies, genetic algorithms, and genetic programming, are 'blind' to constraints. Hence, even if the parents are satisfying some constraints they might very well get offspring violating them. Technically, this means that EAs perform unconstrained search. This observation suggests that EAs are intrinsically unsuited to handle constrained problems.

In this paper we will have a closer look at this phenomenon. We start with describing approaches for handling constraints in evolutionary computation. Next we present an overview of EAs for constraint satisfaction problems, pointing out the key features that have been added to the standard EA machinery in order to handle constraints. In Section 4 we summarize the main lessons learned from the overview and indicate where constraints provide extra information on the problem and how this information can be utilized by an evolutionary algorithm. Thereafter, Section 5 handles a number of methodological considerations regarding research on solving constraint satisfaction problems (CSPs) by means of EAs. The final section concludes this paper reiterating that EAs *are* suited to treat constrained problems and touches on a couple of promising research directions.

2 Constraints handling in EAs

There are many ways to handle constraints in an EA. At a high conceptual level we can distinguish two cases, depending on whether they are handled *indirectly* or *directly*. Indirect constraint handling means that we circumvent the problem of satisfying constraints by incorporating them in the fitness function f such that f optimal implies that the constraints are satisfied, and use the optimization power of the EA to find a solution. By direct constraint handling here we mean that we leave the constraints as they are and 'adapt' the EA to enforce them. We will return later on the differences between these two cases. Let us note direct and indirect constraint handling can be applied in combination, i.e., in one application we can: handle all constraints indirectly; handle all constraints directly, or; handle some constraints directly and others indirectly. Formally, indirect constraint handling means transforming constraints into optimization objectives. The resulting problem transformation imposes the requirement that the (eliminated) constraints are satisfied if the (new) optimization objectives are at their optimum. This implies that the given problem is transformed into an equivalent problem meaning that the two problems share the same solutions[1]. For a given constrained problem several equivalent problems can be defined by choosing the subset of the constraints to be eliminated and/or defining the objective function measuring their satisfaction differently. So, there are two important questions to be answered.

- Which constraints should be handled directly (kept as constraints) and which should be handled indirectly (replaced by optimization objectives)?
- How to define the optimization objectives corresponding to indirectly handled constraints?

Treating constraints directly implies that violating them is not reflected in the fitness function, thus there is no bias towards chromosomes satisfying them. Therefore, the population will not become less and less infeasible w.r.t. these constraints.[2]

[1]Actually, it is sufficient to require that the solutions of the transformed problem are also solutions of the original problem but this nuance is not relevant for this discussion.

[2]At this point we should make a distinction between feasibility in the original problem context and (relaxed) feasibility in the context of the transformed problem. E.g. we could

This means that we have to create and maintain feasible chromosomes in the population. The basic problem in this case is that the regular genetic operators are blind to constraints, mutating one or crossing over two feasible chromosomes can result in infeasible offspring. Typical approaches to handle constraints directly are the following:

- eliminating infeasible candidates,
- repairing infeasible candidates,
- preserving feasibility by special operators,
- decoding, i.e., transforming the search space.

Eliminating infeasible candidates is very inefficient, and therefore hardly applicable. Repairing infeasible candidates requires a repair procedure that modifies a given chromosome such that it will not violate constraints. This technique is thus problem dependent but if a good repair procedure can be developed then it works well in practice, see for instance Section 4.5 in [33] for a comparative case study. The preserving approach amounts to designing and applying problem specific operators that do preserve the feasibility of parent chromosomes. Using such operators the search becomes quasi-free, because the offspring remains in the feasible search space, if the parents were feasible. This is the case in sequencing applications, where a feasible chromosome contains each label (allele) exactly once. The well-known order-based crossovers, [19, 46], are designed to preserve this property. Note that the preserving approach requires the creation of a feasible initial population, which can be NP-hard, e.g., for the traveling salesman problem with time windows. Finally, decoding can simplify the problem and allow an efficient EA. Formally, decoding can be seen as shifting to a search space that is different from the Cartesian product of the domains of the variables in the original problem formulation. Elements of the new search space S' serve as inputs for a decoding procedure that creates feasible solutions, and it is assumed that a free (modulo preserving operators) search can be performed in S' by an EA. For a nice illustration we refer again to Section 4.5 in [33].

In case of indirect constraint handling the optimization objectives replacing the constraints are traditionally viewed as penalties for constraint violation, hence to be minimized. In general penalties are given for violated constraints although some (problem specific) EA allocate penalties for wrongly instantiated variables or, when different from the other options, as the distance to a feasible solution.

Advantages of indirect constraint handling are:

- generality,
- reduction of the problem to ‘simple’ optimization,
- possibility of embedding user preferences by means of weights.

---

introduce the name *allowability* for the conjunction of those constraints that are handled directly. However, to keep the discussion simple we will use the term feasibility for both cases.

Disadvantages of indirect constraint handling are:

- loss of information by packing everything in a single number,
- does not work well for sparse problems,
- how to merge original objective function with penalties?

There are other classification schemes of constraint handling techniques in EC. For instance, the categorization in [32], distinguishes pro-choice and pro-life techniques, where pro-choice encompasses eliminating, decoding, and preserving, while pro-life covers penalty based and repairing approaches. Overviews and comparisons published on evolutionary computation techniques for constraint handling so far mainly concern continuous domains, [29, 30, 31, 34]. Constraint handling in continuous and discrete domains rely to a certain extent on the same ideas. There are, however, also differences, for instance in continuous domains constraints can be characterized as linear, non-linear, etc. and in case of linear constraints special averaging recombination operators can guarantee that offspring of feasible parents are feasible. In discrete domains this is impossible.

The rest of this paper is concerned with a comparative analysis of a number of methods based on EAs for solving CSPs that have been so far introduced. Our comparison is mainly based on the way constraints are handled, either directly or indirectly. Therefore our discussion will not take into account the particular parameters setting of a GA, like the role of mutation and crossover rates, or the role of the selection mechanism and the size of the population. This survey does not pretend to be a comprehensive account of all the works on solving CSP using EAs. It is rather meant to emphasize the main ideas on constraint handling (over finite domains) which have been employed in evolutionary algorithms.

3 Evolutionary CSP solvers

Usually a CSP is stated as a problem of finding an instantiation of variables $v_1, \ldots, v_n$ within the finite domains $D_1, \ldots, D_n$ such that constraints (relations) $c_1, \ldots, c_m$ prescribed for (some of) the variables hold. The formula ϕ is then the conjunction of the given constraints. One may be interested in one, some or all solutions, or only in the existence of a solution.

In the last years there have been reports on quite a few EAs for solving CSPs (for finding one solution) having a satisfactory performance. The majority of these EAs perform indirect constraint handling by means of a penalty based fitness function, and possibly incorporate knowledge about the CSP into the genetic operators, the fitness function, or as apart module in the form of local search. First, we describe four approaches for solving CSPs using GAs that exploit information on the constraint network. Next, we discuss other three methods for solving CSPs which make use of an adaptive fitness function in order to enhance the search for a good (approximate) solution.

3.1 Heuristic Genetic Operators

In [14, 15], Eiben et al. propose to incorporate existing CSP heuristics into genetic operators. Two heuristic based genetic operators are specified: an asexual operator that transforms one individual into a new one and a multi-parent operator that generates one offspring using two or more parents. The asexual heuristic based genetic operator selects a number of variables in a given individual, and then chooses new values for these variables. Both steps are guided by a heuristic: for instance, the selected variables are those involved in the largest number of violated constraints, and the new values for those variables are the values which maximize the number of constraints that become satisfied. The basic mechanism of the multi-parent heuristic crossover operator is scanning: for each position, the values of the variables of the parents in that position are used to determine the value of the variable in that position in the child. The selection of the value is done using the heuristic employed in the asexual operator. The difference with the asexual heuristic operator is that the heuristic does not evaluate all possible values but only those of the variables in the parents. The multi-parent crossover is applied to more parents (typical value 5) and produces one child.

	Version 1	Version 2	Version 3
Main operator	Asexual heuristic operator	Multi-parent heuristic crossover	Multi-parent heuristic crossover
Secondary operator	Random mutation	Random mutation	Asexual heuristic operator
Fitness function	Number of violated constraints		
Extra	None		

Table 1: Specific features of three implemented versions of `H-GA`

The main features of three EAs based on this approach, called `H-GA.1`, `H-GA.2`, and `H-GA.3`, are illustrated in Table 1. In the `H-GA.1` version the heuristic based genetic operator serves as the main search operator assisted by (random) mutation. In `H-GA.3` it accompanies the multi-parent crossover in a role which is normally filled in by mutation.

3.2 Knowledge Based Fitness and Genetic Operators

In [44, 43] M. C. Riff Rojas introduces an EA for solving CSPs which uses information about the constraint network in the fitness function and in the genetic operators (crossover and mutation). The fitness function is based on the notion of *error evaluation* of a constraint. The error evaluation of a constraint is the sum of the number of variables of the constraint and the number of variables that are connected to these variables in the CSP network. The fitness function of an individual, called *arc-fitness*, is the sum of error evaluations of all the violated constraints in the individual. The mutation operator, called *arc-mutation*, selects randomly a variable of an individual and assigns to that variable the value that minimizes the sum of the error-evaluations of the constraints involving that variable. The crossover operator, called *arc-crossover*, selects randomly two parents and builds an offspring by means of the following iterative procedure over all the constraints of the considered CSP. Constraints are ordered according to their error-evaluation with respect to instantiations of the variables that violate the constraints. For the two variables of a selected (binary) constraint c, say v_i, v_j, the following cases are distinguished.

1. If none of the two variables are instantiated in the offspring under construction then:
   - If none of the parents satisfies c, then a pair of values for v_i, v_j from the parents is selected which minimizes the sum of the error evaluations of the constraints containing v_i or v_j whose other variables are already instantiated in the offspring.
   - If there is one parent which satisfies c, then that parent supplies the values for the child.
   - If both parents satisfy c, then the parent which has the higher fitness provides its values for v_i, v_j.
2. If only one variable, say v_i, is not instantiated in the offspring under construction, then the value for v_i is selected from the parent minimizing the sum of the error-evaluations of the constraints involving v_i.
3. If both variables are instantiated in the offspring under construction, then the next constraint (in the ordering described above) is selected.

The main features of a GA based on this approach are summarized in Table 2.

Crossover operator	Arc-crossover operator
Mutation operator	Arc-mutation operator
Fitness function	Arc-fitness
Extra	None

Table 2: Specific features of `Arc-GA`

3.3 Glass Box Approach

In [27] E. Marchiori introduces an EA for solving CSPs which transforms constraints into a canonical form in such a way that there is only one single (type of) primitive constraint. This approach, called glass box approach, is used in constraint programming [48], where CSPs are given in implicit form by means of formulas of a given specification language. For instance, for the N-Queens Problem, we have the well known formulation in terms of the following constraints, where *abs* denotes absolute value:

- $v_i \neq v_j$ for all $i \neq j$ (two queens cannot be on the same row).
- $abs(v_i - v_j) \neq abs(i - j)$ for all $i \neq j$ (two queens cannot be on the same diagonal).

By decomposing complex constraints into primitive ones, the resulting constraints have the same granularity and therefore the same intrinsic difficulty. This rewriting of constraints, called *constraint processing*, is done in two steps: elimination of functional constraints (as in `GENOCOP` [33]) and decomposition of the CSP into primitive constraints. The choice of primitive constraints depends on the specification language. The primitive constraints chosen in the examples considered in [27], the N-Queens Problem and the Five Houses Puzzle, are linear inequalities of the form: $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$. When all constraints are reduced to the same form, a single probabilistic repair rule is applied, called *dependency propagation*. The repair rule used in the examples is of the form **if** $\alpha \cdot p_i - \beta \cdot p_j = \gamma$ **then** change p_i or p_j. The violated constraints are processed in random order. Repairing a violated constraint can result in the production of new violated constraints, which will not be repaired. Thus at the end of the repairing process the chromosome will not in general be a solution. Note that this kind of EA is designed under the implicit assumption that CSPs are given in implicit form by means of formulas in some specification language.

A simple heuristic can be used in the repair rule by selecting the variable whose value has to be changed as the one which occurs in the largest number of constraints, and by setting its value to a different value in the variable domain. The main features of this EA are summarized in Table 3.

Crossover operator	One-point crossover
Mutation operator	Random mutation
Fitness function	Number of violated constraints
Extra	Repair rule

Table 3: Main features of Glass-Box GA

3.4 Genetic local search

In [28] Marchiori and Steenbeek introduced a genetic local search (GLS) algorithm for random binary CSPs, called RIGA (Repair Improve GA). In this approach, heuristic information is not incorporated into the GA operators or fitness function, but is included into the GA as a separate module in the form of a local search procedure. The idea is to combine a simple GA with a local search procedure, where the GA is used to explore the search space, while the local search procedure is mainly responsible for the exploitation. In RIGA, the local search applied to a chromosome produces a consistent partial instantiation, that is, only some of the variables of the CSP have a value, and each constraint of the CSP whose variables are all instantiated dissatisfied. Moreover, this instantiation cannot be extended by binding some non-instantiated variable to a value without violating the consistency. A chromosome is a sequence of actual domains (a subset of the domain), one for each variable of the CSP. RIGA consists of two main phases:

- Repair: a chromosome is transformed into a consistent partial instantiation by removing values from the actual domains of the variables.
- Improve: the consistent partial instantiation is optimized and maximized.

The main features of the GLS algorithm are summarized in Table 4.

Crossover operator	Uniform
Mutation operator	Random mutation
Fitness function	Number of instantiated variables
Extra	Local search

Table 4: Main features of the GLS algorithm

3.5 Co-evolutionary Approach

This approach has been tested by Paredis on different problems, such as neural net learning [39], constraint satisfaction [38, 39] and searching for cellular automata that solve the density classification task [40].

In the co-evolutionary approach for CSPs two populations evolve according to a predator-prey model: a population of (candidate) solutions and a population of constraints. The selection pressure on individuals of one population depends on the fitness of the members of the other population. The fitness of an individual in either of these populations is based on a history of encounters. An *encounter* means that a constraint from the constraint population is matched with a chromosome from the solutions population. If the constraint is not violated by the chromosome, the individual from the solutions population gets a point. Otherwise, the constraint gets a point. The fitness of an individual is the number of points it has obtained in the last 25 encounters. In this way, individuals in the constraint population which have been often violated by members of the solutions population have higher fitness. This forces the solutions to concentrate on more difficult constraints. At every generation of the EA, 20 encounters are executed by repeatedly selecting pairs of individuals from the populations, biasing the selection towards fitter individuals. Clearly, mutation and crossover are only applied to the solutions population. Parents for crossover are selected using linear ranked selection [49]. The main features of this EA are summarized in Table 5.

Another noteworthy example of using a coevolutionary approach to solving satisfaction problems was done by Hisashi Handa et al. in [22, 23]r. Here

Crossover operator	Two-point crossover
Random mutation	
Fitness function	Number of points in last 25 encounters
Extra	Co-evolution

Table 5: Main features of the co-evolutionary algorithm

the host population of solutions competes with a parasite population of useful schemata. These and successive papers explore the use of different operators as well as demonstrate the effectiveness of this kind of coevolutionary approach.

3.6 Heuristic-Based Microgenetic Method

In the approach proposed by Dozier et al in [7], and further refined in [4, 8], information about the constraints is incorporated both in the genetic operators and in the fitness function. In the Microgenetic Iterative Descent Algorithm the fitness function is adaptive and employs Morris' Breakout Creating Mechanism to escape from local optima. At each generation an offspring is created by mutating a specific gene of the selected chromosome, called pivot gene, and that offspring replaces the worst individual of the actual population. The new value for that gene as well as the pivot gene are heuristically selected. 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 is the sum of the weights of all the breakouts[3] whose values occur in the chromosome. 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 [37].

Crossover operator	None
Mutation operator	Singlepoint heuristic mutation
Fitness function	Heuristic based
Extra	None

Table 6: Main features of heuristic-based microgenetic algorithm

In [4, 8], this algorithm is improved by introducing a number of novel features, like a mechanism for reducing the number of redundant evaluations, a novel crossover operator, and a technique for detecting inconsistency.

3.7 Stepwise Adaptation of Weights

The Stepwise Adaptation of Weights (SAW) mechanism has been introduced by Eiben and van der Hauw [11] as an improved version of the weight adaptation

[3]A breakout consists of two parts: 1) a pair of values that violates a constraint; 2) a weight associated to that pair

mechanism of Eiben, Raué and Ruttkay [16, 17]. In several comparisons the SAW-ing EA proved to be a superior technique for solving specific CSPs [2, 12]. The basic idea behind the SAW-ing mechanism is that constraints that are not satisfied after a certain number of steps must be hard, thus must be given a high weight (penalty). The realization of this idea constitutes of initializing the weights at 1 and re-setting them by adding a value δw after a certain period. Re-setting is only applied to those constraints that are violated by the best individual of the given population. Earlier studies indicated the good performance of a simple (1+1) scheme, using a singleton population and exclusively mutation to create offspring. The representation is based on a permutation of the problem variables; a permutation is transformed to a partial instantiation by a simple decoder that considers the variables in the order they occur in the chromosome and assigns the first possible domain value to that variable. If no value is possible without introducing a constraint violation, the variable is left uninstantiated. Uninstantiated variables are, then, penalized and the fitness of the chromosome (a permutation) is the total of these penalties. Let us note that penalizing uninstantiated variables is a much rougher estimation of solution quality than penalizing violated constraints. This option worked well for graph coloring.

Crossover operator	Uniform
Mutation operator	Random mutation
Fitness function	Based on the hardness of constraints
Extra	A decoder to obtain a consistent partial instantiation

Table 7: Main features of the SAW-ing algorithm

4 Discussion

The amount and quality of work in the area of evolutionary CSP solving certainly refutes the initial intuitive hypothesis that EAs are intrinsically unsuited for constrained problems. This raises the question what makes EAs able to solve CSPs? Looking at the specific features of EAs for CSPs one can distinguish two categories. In the first category we find heuristics that can be incorporated in almost any EA component, the fitness function, the variation operators mutation and recombination, the selection mechanism, or used in a repair procedure. The second category is formed by adaptive features, in particular a fitness function that is being modified during a run. All reported algorithms fall into one of these categories and that of Dozier *et al.* belongs to both.

A careful look at the above features discloses that they are all based on information related to the constraints themselves. The very fact that the (global) problem to be solved is defined in terms of (local) constraints to be satisfied facilitates the design and usage of 'tricks'. The scope of applicability of these tricks is limited to constrained problems[4], but not necessarily to a particular CSP, like SAT or graph coloring. The first category of tricks is based on the fact that the presence of constraints facilitates measures on sub-individual structures. For instance, one gene (variable) can be evaluated by the number of conflicts its present value is involved in. Such sub-individual measures are not possible for example in a pure function optimization problem, where only a whole individual can be evaluated. These measures are typically used as evaluation heuristics giving hints on how to proceed in constructing an offspring, or in repairing a given individual. The second category is based on the fact that the composite nature of the problem leads to a composite evaluation function. Such a composite function can be tuned during a run by adding new nogoods (Dozier), modifying weights (SAW-ing), or changing the reference set of constraints used to calculate it (coevolution).

Browsing through the literature there are other aspects that (some of) the papers share. Apparently indirect constraint handling is more common practice than direct constraint handling. On the other hand, in almost all applications some heuristics are used even if the transformed problem is a free optimization problem, and these heuristics are meant to increase the chance of satisfying constraints. In other words, constraints are handled directly by these heuristics.

Another noteworthy property that occurs repeatedly in EAs for CSPs is the small size of the population. Common EA wisdom suggests that big populations are better than small ones for they can keep genetic diversity easier, respectively longer. From personal communications with authors and own experience it turns out that using small populations is always justified by experiments. Exactly because small populations contradict ones intuition, such setups are only taken after substantial experimental justification. Such an experimental comparison sometimes leads to surprising outcomes, for instance that the optimal setup is to use a population of size 1 and only mutation as search operator [2, 10]. In this case it is legitimate to ask whether the resulting algorithm is still evolutionary or is it only just a hill-climber. Clearly, this is a judgment call, but as most people in evolutionary computation accept the (1+1) and the (1,1) evolution strategy as members of the family, it is legitimate to say that one still has an EA in this case.

Summarizing, it seems possible to extract some guidelines from existing literature on how to tackle a CSP by evolutionary algorithms. A short list of promising options is:

1. Use, possibly existing, heuristics to estimate the quality of sub-individual entities (like one variable assignment) in the components of the EA: fitness function, mutation and recombination operators, selection, repair mechanism.

2. Exploit the composite nature of the fitness function and change its composition over time. During the search information is collected (e.g. on

[4]Actually, this is not entirely true. For instance, the SAW-ing technique can be easily imported into GP for machine learning applications, cf. [9]

which constraints are hard); this information can be very well utilized.

3. Try small populations and mutation only schemes.

5 Assessment of EAs for CSPs

The foregoing sections have indicated that evolutionary algorithms can solve constrained problems, in particular CSPs. But are these evolutionary CSP solvers competitive with traditional techniques? Some papers draw a comparison between an EA and another technique, for instance on 3-SAT and graph 3-coloring. In general, however, this question is still open.

Performing an experimental comparison between algorithms, in particular, between evolutionary and other type of problem solvers implies a number of methodological questions:

1. Which benchmark problems and problem instances should be used?

2. Which competitor algorithms should be used?

3. Which comparative measures should be used?

As for the problems and problem instances one could distinguish two main approaches: the repository and the generator approach. The first one amounts to obtaining prepared problem instances that are freely available from (Web-based) repositories, for instance the Constraints Archive at `http://www.cs.unh.edu/ccc/archive`. The advantage of this approach is that the problem instances are 'interesting' in the sense that other researchers have investigated and evaluated them already. Besides, an archive often contains performance reports of other techniques, thereby providing a direct feedback on one's own achievements. Using a problem instance generator (which of course could be coming from an archive) means that problem instances are produced on-the-spot. Such a generator usually has some problem specific parameters, for instance the number of clauses and the number of variables for 3-SAT, or the constraint density and constraint tightness for binary CSPs. The advantage of this approach is that the hardness of the problem instances can be tuned by the parameters of the generator. Recent research has shed light on the location of really hard problem instances, the so-called phase transition, for different classes of problems [5, 20, 21, 24, 35, 41, 42, 45]. A generator makes it possible to perform a systematic investigation in and around the hardest parameter range. The currently available EA literature mostly follows the repository approach tackling commonly studied problems, like N-queens[5], 3-SAT, graph coloring, or the Zebra puzzle. Dozier *et al.* use a random problem instance generator for binary

[5]This problem has a rather exceptional feature: if its size (the number of queens) is increased, it gets easier [36]. This makes it somewhat uninteresting as the traditional 'scale-up competition' won't work with it.

CSPs[6] which creates instances for different constraint tightness and density values [7]. Later on this generator has been adopted and reimplemented by Eiben *et al.* [13].

Advises on the choice for a competitor algorithm boil down to the same suggestion: choose the best one available to represent a real challenge. Implementing this principle is, of course, not always simple. It could be hard to find out which specific algorithm shows the best performance on a given (type of) problem. This is not only due to the difficulties of finding information. Sometimes it is not clear which criteria to use for basing the choice upon.

This problem leads us to the third aspect of comparative experimental research: that of the comparative measures. The performance of a problem solving algorithm can be measured in different ways. Speed and solution quality are widely used, and for stochastic algorithms, as EAs are, the probability of finding a solution (of certain quality) is also a common measure.

Speed is often measured in elapsed computer time, `CPU time` or `user time`. However, this measure is depending on the specific hardware, operating system, compiler, network load, etc. and therefore is ill-suited for reproducible research. In other words, repeating the same experiments, possibly elsewhere, may lead to different results. For generate-and-test style algorithms, as EAs are, a common way around this problem is to count the number of points visited in the search space. Since EAs immediately evaluate each newly generated candidate solution, this measure is usually expressed as the number of fitness evaluations. Forced by the stochastic nature of EAs this is always measured over a number of independent runs and the **A**verage number of **E**valuations to a **S**olution (AES) is used. It is important to note that the average is only taken over the successful runs ("to a Solution"), otherwise the actually used maximum number of evaluations would distort the statistics. Fair as this measure seems, there are two possible problems with it. First, it could be misleading if an EA uses 'hidden labor', for instance some heuristics incorporated in the genetic operators, in the fitness function, or in a local search module (like in GLS). The extra computational effort due to hidden labor can increase performance, but are invisible to the AES measure[7]. Second, it can be difficult to apply AES for comparing an EA with search algorithms that do not work in the same search space. An EA is iteratively improving complete candidate solutions, so one elementary search step is the creation of one new candidate solution. However, a constructive search algorithm would work in the space of partial solutions (including the complete ones that an EA is searching through) and one elementary search step is extending the current solution. Counting the number of elementary search steps is misleading if the search steps are different. A common treatment for both of these problems with AES (hidden labor, different search steps) is to

[6]Binary CSPs (where each constraint concerns exactly two variables) form a nice problem class. While they have a transparent structure it holds that every CSP is equivalent to a binary CSP [47].

[7]In the CSP literature the number of constraint checks is used commonly as speed measure. It seems an interesting option to use this into measure in combination with or as an alternative to the AES measure in evolutionary computing

compare scale-up behavior of the algorithms. To this end a problem is needed that is scalable, that is, its size can be changed. The number of variables is a natural scale-up parameter for many problems. Two different types of methods can then be compared by plotting their own speed measure figures against the problem size. Even though the measures used in each curve are different, the steepness information is a fair basis for comparison: the curve that grows at a higher rate indicates an inferior algorithm.

Solution quality of approximate algorithms for optimization is most commonly defined as the distance to an optimum at termination, e.g. $|f_{best} - f_{opt}|$, where f is the function to be optimized, f_{best} is the f value of best candidate solution found in the given run and f_{opt} is the optimal f value. For stochastic algorithms this is averaged over a number of independent runs and in evolutionary computing the **M**ean **B**est **F**itness (MBF) is a commonly used name for this measure. As we have seen in this paper, for constraint satisfaction problems it is not straightforward what f to use – there are more sensible options. For comparing the solution quality of algorithms this means that there are more sensible quality measures. The problem is then, that most probably one would use the function f that has been used to find a solution and this can be different for another algorithm. For instance, algorithm A could use the number of unsatisfied constraints as fitness function and algorithm B could use the number of wrong variable instantiations. It is then not clear what measure to use for comparing the two algorithms. Moreover, in constraint satisfaction it is often not good enough to be close to a solution. A candidate is either good (satisfies all constraints) or bad (violates some constraints). In this case, it makes no sense to look at the distance to a solution as a quality measure, hence the MBF measure is not appropriate.

The third measure which is often used to judge stochastic algorithms, and thus EAs, is the probability of finding a solution (of certain quality). This probability can be estimated by performing a number of independent runs under the same setup on the same type of problems and keep a record on the percentage of runs that did find a solution. This **S**uccess **R**ate (SR) completes the picture obtained by AES and MBF. Note that SR and MBF are related but do provide different information, and all different combinations of good/bad SR/MBF are possible. For instance, bad (low) SR and good (high) MBF indicate a good approximator algorithm: it gets close, but misses the last step to hit the solution. Likewise, a good (high) SR and a bad (low) MBF combination is also possible. Such a combination shows that the algorithm mostly performs perfectly, but sometimes it does a very very bad job.

6 Conclusion

This survey of related work disclosed how EAs can be made successful in solving CSPs. Roughly classifying the options we encountered, the key features are the utilization of heuristics and/or the adaptation of the fitness function during a run. Both features are based on the structure of the problems in question, so in a way the problem of how to treat CSPs carries its own solution.

In particular, constraints facilitate the use of sub-individual measures to evaluate parts of candidate solutions. Such sub-individual measures are not possible for example in a pure function optimization problem, where only a whole individual can be evaluated. These measures lead to heuristics that can be incorporated in practically any component of an EA, the fitness function, mutation and recombination operators, selection, or used in a repair (or more in general local search) mechanism.

Likewise, it is the presence of constraints that leads to a fitness function composed from separate pieces. This composition or the relative importance of the components can be changed over time. During the search information is collected (e.g. on which constraints are hard) and this information can be very well utilized.

The field of evolutionary constraint satisfaction is relatively new. Intensive investigations started approximately in the mid nineties, while evolutionary computing itself has it roots in the sixties. Because of the short history coherence is lacking and the findings of individual experimental studies cannot be generalized (yet). There are a number of research directions that should be pursued in the future for further development. These include:

- Study of the problem area. A lot can be learned from the traditional constrained literature about such problems. Existing knowledge should be imported into core EC research.

- Cross-fertilization between the insights concerning EAs for (continuous) COPs and (discrete) CSPs. At present, these two sub-areas are practically unrelated.

- Sound methodology: how to set up fair experimental research, how to obtain good benchmarks, how to compare EAs with other techniques.

- Theory: better analysis of the specific features of constrained problems, and the influence of these features on EA behavior.

References

[1] Th. Bäck, editor. *Proceedings of the 7th International Conference on Genetic Algorithms*, San Francisco, CA, 1997. Morgan Kaufmann Publishers, Inc.

[2] Th. Bäck, A.E. Eiben, and M.E. Vink. A superior evolutionary algorithm for 3-SAT. In V.W. Porto, N. Saravanan, D. Waagen, and A.E. Eiben, editors, *Proceedings of the 7th Annual Conference on Evolutionary Programming*, number 1477 in Lecture Notes in Computer Science, pages 125–136, Berlin, 1998. Springer-Verlag.

[3] R.K. Belew and L.B. Booker, editors. *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc., 1991.

[4] J. Bowen and G. Dozier. Solving constraint satisfaction problems using a genetic/systematic search hybride that realizes when to quit. In Eshelman [18], pages 122–129.

[5] P. Cheeseman, B. Kenefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th IJCAI*, pages 331–337. Morgan Kaufmann Publishers, Inc., 1991.

[6] A.G. Cohn, editor. *Proceedings of the 11th European Conference on Artificial Intelligence*, New York, NY, 1994. John Wiley & Sons.

[7] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In IEEE [25], pages 306–311.

[8] G. Dozier, J. Bowen, and D. Bahler. Solving randomly generated constraint satisfaction problems using a micro-evolutionary hybrid that evolves a population of hill-climbers. In *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, pages 614–619. IEEE Computer Society Press, 1995.

[9] J. Eggermont, A.E. Eiben, and J.I. van Hemert. Adapting the fitness function in GP for data mining. In R. Poli, P. Nordin, W.B. Langdon, and T.C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *Lecture Notes in Computer Science*, pages 195–204. Springer-Verlag, 1999.

[10] A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In IEEE [26], pages 81–86.

[11] A.E. Eiben and J.K. van der Hauw. Adaptive penalties for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution '97*, number 1363 in Lecture Notes in Computer Science, pages 95–106, Berlin, 1998. Springer-Verlag.

[12] 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.

[13] A.E. Eiben, J.I. van Hemert, E. Marchiori, and A.G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in Lecture Notes in Computer Science, pages 196–205, Berlin, 1998. Springer-Verlag.

[14] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Heuristic genetic algorithms for constrained problems, part i: Principles. Technical Report IR-337, Vrije Universiteit Amsterdam, 1993.

[15] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In IEEE [25], pages 542–547.

[16] 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.

[17] 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 Computer Society Press, 1996.

[18] L.J. Eshelman, editor. *Proceedings of the 6th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc., 1995.

[19] B.R. Fox and M.B. McMahon. Genetic operators for sequencing problems. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 284–300. Morgan Kaufmann Publishers, Inc., 1991.

[20] I. Gent, E. MacIntyre, P. Prosser, and T. Walsh. Scaling effects in the CSP phase transition. In U. Monanari and F. Rossi, editors, *Principles and Practice of Constraint Programming — CP95*, Berlin, 1995. Springer-Verlag. also available at `http://www.cs.strath.ac.uk/~apes/apepapers.html`.

[21] I. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions*. IOS Press, 1995.

[22] H. Handa, N. Baba, O. Katai, T. Sawaragi, and T. Horiuchi. Genetic algorithm involving coevolution mechanism to search for effective genetic information. In IEEE [26].

[23] H. Handa, C. O. Katai, N. Baba, and T. Sawaragi. Solving constraint satisfaction problems by using coevolutionary genetic algorithms. In *Proceedings of the 5th IEEE Conference on Evolutionary Computation*, pages 21–26. IEEE Computer Society Press, 1998.

[24] T. Hogg and C. Williams. The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69:359–377, 1994.

[25] *Proceedings of the 1st IEEE Conference on Evolutionary Computation*. IEEE Computer Society Press, 1994.

[26] *Proceedings of the 4th IEEE Conference on Evolutionary Computation*. IEEE Computer Society Press, 1997.

[27] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In Bäck [1], pages 330–337.

[28] E. Marchiori and A. Steenbeek. Genetic local search algorithm for random binary constraint satisfaction problems. In *Proceedings of the ACM Symposium on Applied Computing*, 2000. to appear.

[29] Z. Michalewicz. Genetic algorithms, numerical optimization, and constraints. In Eshelman [18], pages 151–158.

[30] Z. Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. In J.R. McDonnell, R.G. Reynolds, and D.B. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*, pages 135–155, Cambridge, MA, 1995. MIT Press.

[31] Z. Michalewicz and N. Attia. Evolutionary optimization of constrained problems. In A.V. Sebald and L.J. Fogel, editors, *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, pages 98–108. World Scientific, 1994.

[32] Z. Michalewicz and M. Michalewicz. Pro-life versus pro-choice strategies in evolutionary computation techniques. In M. Palaniswami, Y. Attikiouzel, R.J. Marks, D. Fogel, and T. Fukuda, editors, *Computational Intelligence: A Dynamic System Perspective*, pages 137–151. IEEE Computer Society Press, 1995.

[33] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[34] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Journal of Evolutionary Computation*, 4(1):1–32, 1996.

[35] D. Mitchell, B. Selman, and H.J. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI-92*, pages 459–465. AAAI Press/The MIT Press, 1992.

[36] P. Morris. On the density of solutions in equilibrium points for the n-queens problem. In *Proceedings of the 9th International Conference on Artificial Intelligence (AAAI-92)*, pages 428–433, 1992.

[37] 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.

[38] J. Paredis. Coevolutionary constraint satisfaction. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in Lecture Notes in Computer Science, pages 46–55, Berlin, 1994. Springer-Verlag.

[39] J. Paredis. Co-evolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[40] J. Paredis. Coevolving cellular automata: Be aware of the red queen. In Bäck [1].

[41] P. Prosser. Binary constraint satisfaction problems: Some are harder than others. In Cohn [6], pages 95–99.

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

[43] M.C. Riff-Rojas. Evolutionary search guided by the constraint network to solve CSP. In Belew and Booker [3], pages 337–348.

[44] M.C. Riff-Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves CSP. In Belew and Booker [3], pages 279–284.

[45] B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. In Cohn [6], pages 100–104.

[46] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic sequenceing operators. In Belew and Booker [3], pages 69–76.

[47] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 1993.

[48] P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). In A. Podelski, editor, *Constraint Programming: Basics and Trends*. Springer-Verlag, Berlin, 1995.

[49] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 116–123, San Mateo, California, 1989. Morgan Kaufmann Publishers, Inc.