An Experimental Comparison of Three Different Heuristic GAs for Solving Constraint Satisfaction Problems

B.G.W. Craenen

November 2, 1999

Abstract

In this thesis three different techniques for solving Constraint Satisfaction Problems (CSPs) with Genetic Algorithms (GAs) are compared on a set of benchmark problems consisting of randomly generated binary constraint satisfaction problems. The techniques that are investigated exploit heuristic information on the constraint network to help traditional GAs solve CSPs. Implemented are three different GAs using these techniques: `ESP-GA`, which uses a constraint processing phase and a probabilistic repair rule by E. Marchiori; `H-GA`, using heuristic genetic operators by Eiben et al.; and `Arc-GA`, which uses two new genetic operators by M. C. Riff Rojas and a new fitness function that are guided by information from the constraint network. Three different versions of `H-GA` were evaluated. These three GAs are tested on benchmark problems obtained by a random generator of binary CSP instances, which generates constraints whose density and tightness can be specified by the user. This allows one to study the performance of these algorithms on different kinds of CSPs. Although the results of the different GAs lie close together, they show that two versions of `H-GA` perform slightly better than the third version of `H-GA` and `ESP-GA` with `Arc-GA` performing the least when comparing success rates. This seems to support the notion that GAs using strong heuristics are prone to premature convergence while GAs with more general heuristics are still able to escape local optima. Finally, some options for future study are explored.

Contents

List of Tables

List of Figures

1 Introduction

Constraint satisfaction has become a research topic common to many different research communities (cf. e.g. [3, 16, 19, 34]), due to its practical relevance in many application areas like operations research, hardware design and artificial intelligence. In particular, in the last couple of years, various techniques based on genetic algorithms have been developed for the solution of CSPs [4, 7, 11, 23, 26, 27]. Moreover, search heuristics and constraint propagation techniques developed in previous works on constraint processing (cf., [34, 35]) have been incorporated into genetic operators [8, 26, 29]. Due to its stochastic nature, a GA does not provide a complete tool for solving CSPs, and in general is not used for detecting whether a CSP is unsatisfiable. A notable exception is a recent proposal [1] that has introduced a hybrid algorithm that combines concepts from genetic algorithms and hill climbing, and that incorporates arc revision in order to decide when to stop (with failure) in case the CSP is unsatisfiable.

Usually GAs are considered to be ill-suited for solving CSPs. This is because the traditional search operators (mutation and recombination) are considered to be 'blind' to the constraints of the problem. As the traditional search operators do not take into account the variable interdependencies that are inherent in CSPs, applying them to an individual can result in the violation of a constraint where this was not the case before the application. Another reason why GAs are considered to be ill-suited to solving CSPs is the absence of an objective function in a CSP — there is only a set of constraints to be satisfied — while traditionally an GA is used to optimize. Despite such general arguments, in recent years, there have been reports on quite a few GAs for solving CSPs that have a satisfactory performance.

Roughly speaking, these GAs can be divided into two categories: those using a fitness or penalty function that is adapted during the search [1, 4, 5, 10, 12, 13, 14, 27, 28], and those based on exploiting heuristic information on the constraint network [7, 8, 9, 20, 31, 32]. In this thesis three methods are investigated from the second category: the process and repair method by E. Marchiori [20], the method using heuristic genetic operators by Eiben et al. [7], and the GA using genetic operators and a fitness function that are guided by the constraint network by M. C. Riff Rojas [31, 32]. Three specific GAs are implemented based on these corresponding methods, called ESP-GA, H-GA, and Arc-GA respectively, in this thesis. All three implementations were compared on a test suite consisting of randomly generated binary CSPs with finite domains.

Although the three algorithms I used were firmly based on the original algorithms, some adaptation were necessary. First, I had to translate the methods that were set out in the original articles into algorithms that worked well within the library I used. Secondly, some alterations were necessary to perform a comparative analysis, making the algorithms work together with the CSP instance generator is one of them. Finally I have made some design choices regarding the heuristics that the algorithms had to use. I have used relatively the same heuristics where this was possible, again, this was done to perform a fair comparative analysis between the different GAs. Furthermore, I have tried to put together a framework of terms and definitions early in the thesis, to bridge the possible gap in terminology that can occur when two different fields of study are discussed which both have similar terms for different concepts.

In order to study the relative performance of these algorithms I use a set of benchmark problems consisting of randomly generated problem instances, where the hardness of the problem instances is influenced by two parameters: constraint tightness and constraint density. Detailed feedback on GA behavior is gained by running experiments on 25 different combinations of these parameters. While considering 10 runs on 250 problem instances, I can summarize that GAs that use general heuristics had the best success rate of all three GAs. The first two versions of H-GA fall in this category. GAs with strong heuristics, like Arc-GA, seem less able to escape local optima and have a success rate that is lower than the other GAs. ESP-GA, using a dependency propagation or repair rule, has a performance in between these two GAs.

The thesis is organized as follows: the next section describes the notion of constrained problems and introduces some standard definitions. It also deals with the random CSP instance generator that I use in my experiments and the output it produces. After that, some general notions about solving CSPs with GAs are mentioned. In section 3 the adaptation of the method by E. Marchiori to the random CSP instance generator and its output is explained. In section 4 the same is done with the method of Eiben et al. as is with the GA from M. C. Riff Rojas in section 5. In section 6 the exact software description and experimental setup of all three methods is given. The next section (section 7), gives the results of the experiments while in section 8 these results are compared. After that, in section 9, my conclusions are drawn, and finally, in section 10, I discuss some possibilities for future study.

2 Constraint satisfaction problems and GAs

2.1 Constraint satisfaction problems

Constrained problems can be roughly divided into two categories: constrained optimization problems and constraint satisfaction problems [11].

A *constrained optimization problem* (COP) is a triple $\langle S, f, \phi \rangle$, where S is a free search space (i.e. $S = D_1 \times \ldots \times D_n$ is a Cartesian product of sets), f is a (real valued) objective function on S and ϕ is a formula (Boolean function on S). A *solution of a constrained optimization problem* is an $s \in S$ with $\phi(s) = \mathit{true}$ and an optimal f-value.

A *constraint satisfaction problem* (CSP) is a pair $\langle S, \phi \rangle$, where S is a free search space and ϕ is a formula (Boolean function on S). A *solution of a constraint satisfaction problem* is an $s \in S$ with $\phi(s) = \mathit{true}$.

Usually a CSP is stated as a problem of finding a set of values $p_1, \ldots, p_n$ of variables $v_1, \ldots, v_n$ within the finite domains $D_1, \ldots, D_n$ such that constraints $c_1, \ldots, c_k$ hold. I use p_i to indicate the values, v_i for variables, D_i for the domains over these variables, n for the total number of variables and k for the total number of constraints. 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 this thesis I restrict the discussion to finding one solution. With this terminology, solving a CSP means finding one feasible element of the search space while solving a COP means finding a feasible and optimal element. Solving COPs by GAs is extensively treated in [21, 22] and [25], the present investigation concerns solving CSPs by GAs.

In this thesis *binary constraint satisfaction problems* over finite domains are considered, this means that constraints act between pairs of variables. This is not restrictive however since any CSP can be reduced to a binary CSP by using a suitable transformation which involves the definition of more complex domains (cf. [34]).

Because we look at binary constraint satisfaction problems a *constraint* can be written like:

$$c : D_i \times D_j \to \{0, 1\}$$

I adopt here the convention that $c(p_i, p_j) = 1$ if $c(p_i, p_j)$ is false, that is if the value pair $\langle p_i, p_j \rangle$ violates c. Without lose of generality one can assume that for each constraint $(i < j)$ holds[1]. Sometimes it is useful to indicate the used constraint by its (two) variables, in that case I use the notation $c_{i,j}$.

A constraint $c : D_i \times D_j \to \{0, 1\}$ is *relevant* to variables v_i and v_j and is not relevant to other variables. The *cardinality* with respect to a CSP of a variable is the number of constraints relevant to that variable. *Satisfied cardinality* is the number of satisfied constraints relevant to the variable while when calculating the *unsatisfied cardinality* , only the unsatisfied constraints relevant to the variable are counted.

In order to extract information about the constraints of the CSP and the relation they have to each other, I use a *constraint matrix* . A constraint matrix belonging to a CSP is a $R = k \times n$ matrix whose (i, j)-th element is 1 when v_j is relevant for c_i[2]:

$$R_{i,j} = \begin{cases} 1 & \text{if variable } v_j \text{ is relevant for } c_i \\ 0 & \text{otherwise} \end{cases}$$

It is convenient to distinguish two classes of (binary) constraints, called functional and relational. *Functional constraints* are such that for every pair of solutions $\langle p_1, p_2 \rangle$ and $\langle p'_1, p'_2 \rangle$, if $p_1 = p'_1$ then $p_2 = p'_2$. A constraint that is not functional is called a *relational constraint.*

A *conflict* belonging to constraint c is a value pair $\langle p_i, p_j \rangle \in D_i \times D_j$ $(i < j)$ such that $c(p_i, p_j) = 1$, that is, a conflict is a value pair that violates a constraint.

[1]If this was not the case, one can 'merge' constraint $c : D_i \times D_j \to \{0, 1\}$ and $d : D_j \times D_i \to \{0, 1\}$ into one constraint $e : D_i \times D_j \to \{0, 1\}$ in such a way that $e(p_i, p_j) = 1 \Leftrightarrow c(p_i, p_j) = 1$ or $d(p_j, p_i) = 1$.

[2]Because in this thesis I talk about binary CSPs, of the entries bearing on a constraint, only two are non-zero as there are only two relevant variables to every constraint.

The *set of conflicts* belonging to constraint $c : D_i \times D_j \rightarrow \{0, 1\}$ can then be defined as:

$$\{\langle p_i, p_j \rangle \in D_i \times D_j | c(p_i, p_j) = 1\}$$

A *conflict matrix* belonging to constraint $c : D_i \times D_j \rightarrow \{0, 1\}$ is a $|D_i| \times |D_j|$ matrix whose (i, j)-th element is 0 iff $c(p_i, p_j) = 0$ and 1 iff $c(p_i, p_j) = 1$:

$$C_{i,j} = \begin{cases} 1 & \text{iff } c(p_i, p_j) = 1 \\ 0 & \text{iff } c(p_i, p_j) = 0 \end{cases}$$

An *arc* is a pair of variables belonging to a constraint $c : D_i \times D_j \rightarrow \{0, 1\}$:

$$\langle v_i, v_j \rangle \text{ with } (i < j)$$

Because an arc specifies only part of a constraint, *arc-relevance* and *arc-cardinality* can be defined in a similar way as was earlier done with variable-relevance and variable-cardinality[3].

The *constraint network* can be defined as an undirected graph $G = (N, E)$ where N is the set of variables and E is the set of arcs that connect these variables.

It is also useful to define instantiations. A *complete instantiation* is a mapping $(v_1, \ldots, v_n) \rightarrow D_1 \times \ldots \times D_n$. A *partial instantiation* is a mapping from some of the variables such that it assigns to each of the mapped variables a value from its domain. A *violator instantiation* is an instantiation (partial or complete) with the additional requirement that the instantiation violates at least one constraint, in other words, its values contain a conflict with one of the constraints.

A class of random binary CSPs can be specified by four parameters $\langle n, m, d, t \rangle$, where n is the number of variables, m is the uniform domain size (i.e. $|D_i| = m$ for $i = 1, \ldots, n$), d is the probability that a constraint exists between two variables (*constraint density*) and t is the probability of a conflict between two values along a given constraint (*constraint tightness*).

CSPs exhibit a *phase transition* when one of these parameters 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 solvable, changes from almost one to almost zero. Within this mushy region, problems are in general difficult to solve or to prove unsolvable. An important issue in the study of binary CSPs is to identify those problem instances which are very hard to solve [2]. Recent theoretical investigations ([33, 37]) allow one to predict where the hardest problem instances should occur. Williams and Hogg in [37] develop a theory that predicts that the phase transition occurs when per variable

[3]Because an arc only specifies the pair of variables to a constraint, and it does not specify which pair of variables is in conflict or not, one cannot determine if an arc, as such, is satisfied or not. Therefore, one cannot determine the satisfied or unsatisfied cardinality of an arc whereas that was possible with constraints.

there are a critical number of nogoods (i.e., of conflicts between that variable and all others)[4]. Smith in [33] conjectures that the phase transition occurs when problems have, on average, just one solution.

An experimental investigation with a complete algorithm (i.e., an algorithm that finds a solution or detects unsatisfiability) based on forward checking and on conflict-directed backjumping, is given by Prosser in [30], which provides empirical support to the theoretical prediction given in [33, 37] for higher density/tightness of the constraint networks.

2.2 The random `CSP` instance generator

To generate a test suite, a `CSP` *instance generator* was used, which was developed by J.I. van Hemert, loosely based on the instance generator of G. Dozier [1]. Given $\langle n, m, d, t \rangle$ the instance generator first calculates the number of constraints that will be produced using the following equation:

$$\text{Number of constraints } = \frac{n(n-1)}{2} \cdot d$$

It then starts producing constraints by randomly choosing two variables and defining a constraint between them. When a constraint is defined between variable v_i and v_j, a conflict matrix of conflicting values is generated. The number of conflicts in this table is determined in advance by this equation:

$$\text{Number of conflicts } = m^2 \cdot t$$

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 random `CSP` instance generator produces output in the following form: The first two lines of the output indicate the number of variables (n) and the domain size of each variable (m). In this thesis all domains have an equal size, so in the input used in this thesis these values are the same. After a blank line, a column of zeros and ones represents the conflict matrices between the value pairs of the variables of the `CSP`.

Suppose there are n different variables, all having a domain size of m, the first table of $m \times m$ entries defines the conflicts of the first variable (v_1) with the second variable (v_2). The next $m \times m$ table defines the conflicts between variables (v_1) and (v_3) etc. until all constraints that are relevant to the first variable have been defined by their conflict matrix. After this, the constraints of the second variable (v_2) with the third variable (v_3) are defined until finally all constraints between all variables have been set. If there's no constraint between two variables, the array contains only zeros.

In appendix A a sample output of the `CSP` instance generator is given. All remarks not in 'type style' are edited in later to give better understanding of the contents of the output. Some tables of this output will be used as an ongoing

[4]The expected number of nogoods per variable is $dtm^2(n-1)$.

example throughout this thesis. The sample defines a five variable CSP with an equal domain size of five and a constraint density of 0.5 and a constraint tightness of 0.5.

2.3 Constraint Satisfaction Problems and Genetic Algorithms

In this thesis I will — and have already — use many terms that originate from CSP-terminology. In table 1, I have put together, mostly for clarity, the synonyms for these terms for readers who are more familiar with the classic GA-terminology. I use the CSP-terminology mostly because I feel that it suites best the combined fields of GAs and CSPs.

Constraint Solving Problems	**Evolutionary Algorithms**
Complete Instantiation	Chromosome, Individual
Variable	Gene
Value	Allele

Table 1: Table of CSP - GA synonyms

There are several ways to handle constraints in an GA. At a high conceptual level two cases can be distinguished, depending on whether the constraints are handled: *indirectly* or *directly* [11]. Indirect constraint handling means that the problem of satisfying constraints is circumvented by incorporating them in the fitness function f such that the optimality of f implies that the constraints are satisfied. Then the optimization power of the GA can be used to find a solution. By direct constraint handling I mean that the constraints are left as they are and 'something' is done in the GA to enforce them. Some commonly used options are repair mechanisms, decoding algorithms and using special reproduction operators [11, 24].

In this thesis I pay attention to GAs that use direct handling to find a solution to CSPs. ESP-GAs use direct handling because they repair the individuals. Repairing the individuals is done based on the propagation of the dependencies of the variables of the CSP. To make this possible, some precalculation of the CSP is necessary. H-GAs use new crossover or mutation methods that incorporate heuristics to find better individuals while Arc-GAs use next to new reproduction operators also a new fitness evaluation system, which all base their choices on a search of the constraint network. All three methods use direct handling of the constraints as a way to generate solutions to the CSP.

3 Solving CSPs with ESP-GAs

In [20], E. Marchiori suggests a new approach to solving CSPs by GAs. The approach consists of two main phases in the design of the GA. In the first phase the constraints of the CSP are rewritten and in the second phase the CSP is solved by a GA with an embedded repair rule. In her article she names the method ESP-GA after: Elimination, Splitting and Propagation. The method has been tested on the *five-houses puzzle* and the *n-queens problem* were it had satisfactory performance, even when not all steps of the method were used.

The idea is based on the '*glass-box*' approach [36] because it adjusts the CSP in such a way that there is only one single (type of) primitive constraint. By decomposing more complex constraints into primitive ones, the resulting constraints have the same granularity and therefore the same intrinsic difficulty.

This rewriting of constraints is done in two steps and is called *constraint processing*. Because after the constraints are rewritten, all constraints have an equal form, a single repair rule can be used in the GA to enforce *dependency propagation*. Because all constraints share a single repair rule, repairing an individual can be performed locally by applying the repair rule to every violating constraint.

As said earlier, the method proposed by E. Marchiori consists of two phases: The first phase, called *constraint processing*, rewrites the original CSP in two steps: First, called the *elimination step*, functional constraints are eliminated in order to reduce the number of variables in the problem. This is done analogously to the operation used, e.g., in GENOCOP [23]. In the second step, called the *splitting step*, the resulting constraints are decomposed into a set of constraints in canonical form, a composition of primitive constraints. The constraints proposed are of the form[5]:

$$\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$$

Because some of the variables are discarded during the elimination of functional constraints, these have to be recovered when the GA yields a solution. In this way a solution of the original CSP can be calculated.

In the second phase, called *dependency propagation*, the adjusted CSP is solved using a GA that incorporates a form of probabilistic *repair rule*. It deals with violations of primitive constraints.[6] The repair rule proposed is of the form:

$$\textbf{if } \alpha \cdot p_i - \beta \cdot p_j = \gamma \textbf{ then } \text{modify } p_i \text{ or } p_j$$

Summarizing, the following scheme of the algorithm can be obtained:

1. rewriting the CSP using constraint processing:
    - (a) elimination step: eliminate functional constraints
    - (b) splitting step: decompose the remaining constraints into constraints in one single canonical form
2. Solving the CSP using a GA with dependency propagation in the form of a repair rule

[5]other constraint forms are also possible

[6]because all constraints of the CSP, after constraint processing, are of the same canonical form, only one single repair-rule is needed

3.1 Constraint processing

The *implementation of constraint processing* is done by converting the conflict tables that were produced by the `CSP` instance generator into constraints in the form that was proposed by E. Marchiori:

$$\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$$

Converting the conflict tables into this type of constraint can be done by multiplying the value of the first variable (v_i) with the domain size of the second variable and then subtract the value of the second variable:

$$\gamma = |D_j| \cdot v_i - v_j$$

This simply means that the γ-value of the proposed constraint is calculated by taking $\alpha = D_j$ and $\beta = 1$. To check violation of a constraint of this form one enters the values for the specific variables. If the result is the calculated γ-value, the constraint is violated. A small example: suppose two variables v_2 and v_3 with a uniform domain size of five and with the following conflict matrix:

$$\begin{array}{cc} & v_3 \\ v_2 & \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

The array can be recalculated in the explained way into the following table of γ-values:

$$\begin{array}{cc} & v_3 \\ v_2 & \begin{pmatrix} 4 & 3 & 2 & 1 & 0 \\ 9 & 8 & 7 & 6 & 5 \\ 14 & 13 & 12 & 11 & 10 \\ 19 & 18 & 17 & 16 & 15 \\ 24 & 23 & 22 & 21 & 20 \end{pmatrix} \end{array}$$

Because I am only interested in the conflicts, only the γ-values which had a conflict (the non-zero entries in the conflict matrix) have to be stored. When replacing the non-conflict entries of the earlier table with '$-$'-labels, the following matrix of γ-values is obtained:

$$\begin{array}{cc} & v_3 \\ v_2 & \begin{pmatrix} 4 & - & 2 & - & 0 \\ - & 8 & - & - & 5 \\ - & 13 & 12 & - & - \\ 19 & - & - & - & - \\ - & - & 22 & 21 & - \end{pmatrix} \end{array}$$

The resulting constraints in the form as proposed by E. Marchiori are given below:

$$\begin{array}{ll}
5 \cdot v_2 - v_3 \neq 0 & 5 \cdot v_2 - v_3 \neq 12 \\
5 \cdot v_2 - v_3 \neq 2 & 5 \cdot v_2 - v_3 \neq 13 \\
5 \cdot v_2 - v_3 \neq 4 & 5 \cdot v_2 - v_3 \neq 19 \\
5 \cdot v_2 - v_3 \neq 5 & 5 \cdot v_2 - v_3 \neq 21 \\
5 \cdot v_2 - v_3 \neq 8 & 5 \cdot v_2 - v_3 \neq 22
\end{array}$$

I prove that the above transformation produces an equivalent CSP' from the original CSP [7]. First, notice that for every constraint $c_{i,j}$ in CSP and for every $\langle p_r, p_s \rangle$ which violates $c_{i,j}$ the constraint

$$c'_{i,j} \equiv |D| \cdot v_i - v_j \neq |D| \cdot p_r - p_s$$

is in CSP'. Thus $\langle p_r, p_s \rangle$ violates $c'_{i,j}$. So we have shown that if CSP is violated, CSP' is also violated. Vice-versa, suppose $c'_{i,j}$ in CSP' is violated by the assignment $\langle p'_r, p'_s \rangle$. Then

$$|D| \cdot p'_r - p'_s = |D| \cdot p_r - p_s,$$

$$|D| \cdot (p'_r - p_r) = p'_s - p_s.$$

But, by the way I defined D, we have $D = \{1, 2, \cdots, |D|\}$. Thus if $p'_r \neq p_r$ then $|p'_s - p_s| \geq |D|$ which is impossible, so $p'_r = p_r$ and $p'_s = p_s$, so $\langle p'_r, p'_s \rangle$ violates CSP'.

The technique, as explained above, can be used with every domain size and is also useful for CSPs with non-uniform domain sizes .

It is interesting to note that, with the technique as explained above, it is not necessary to apply the elimination and splitting step as was proposed by E. Marchiori. This is mostly due to the random CSP instance generator and the already simplified format of its output. If more traditional CSPs were involved, like the *n-queens problem* and the *five-houses puzzle* that were used by E. Marchiori, the two steps of constraints processing is useful. The question remains however if rewriting these problems into the format that the CSP instance generator uses as output — after which a computer can apply the above mentioned technique to rewrite the constraints — is not more efficient than using the two constraint processing steps that must be done by hand.

Another interesting note is that all conflicts in the binary CSP are translated into one constraint each. This means that for a CSP with a large number of conflicts — problems with high constraint tightness and/or density — a large number of constraints have to be satisfied. The type of constraint however requires a low execution time to check and introduces only linear overhead to the original problem. The question remains however if the total overhead of an

[7] i.e., if a complete assignment of values satisfies CSP', then it also satisfies the original CSP, and vice-versa

actual GA trying to solve a CSP with high constraint tightness and/or density does not slow the solving process down too much. The translation method proposed however uses just a little storage-space as only the γ-values of the conflicts (constraints) have to be stored and only simple calculation-steps are necessary to check these constraints. This should ensure an acceptable execution-time.

3.2 Dependency propagation

When all the constraints are converted into the chosen form, the repair rule for the *implementation of dependency propagation* in the GA, can be chosen. E. Marchiori proposes the following repair rule:

$$\textbf{if } \alpha \cdot p_i - \beta \cdot p_j = \gamma \textbf{ then } \text{modify } p_i \text{ or } p_j$$

In [20], the repair rule is tested as quite effective when the choice of the variable to be changed was done randomly, I however propose a variation on this by using a simple and straightforward bias system to determine which of the two values (p_i or p_j) should be modified. In general there are three issues to the repair rule that should be addressed.

3.2.1 Selecting the variable

The first issue is, *selecting the variable* that should be repaired. This issue can be resolved by looking at the conflict matrix. The variable with the smallest number of zeros on its row or column is the most restricted, simply because there are more values that result in a conflict if there are just a few zeros compared to a value that has many zeros on its row or column. Counting the zeros in the conflict matrix for every value of the two variables is enough to determine which one of the variables is to be changed. With an equal number of zeros, a random choice is made. By changing the most restricted variable I hope to improve individuals with variables that are hard to satisfy, early on in the process. By repairing the most restricted variable I also hope to find solutions even when the problem has a high constraint tightness and/or density.

3.2.2 Selecting the value

The second issue is *selecting the value* that the variable should be changed to. Because I know the number of feasible values that the variable has, I can make a random selection among these values. By making a random selection at this point, I hope, in some way, to counter the greedy selection of the variable. Note that, although the choice is random, I make sure that only a value that introduces no new conflict can be chosen, thus ensuring that a conflict before the repair rule was applied to the individual is not replaced by another one, furthermore, the old value of the selected variable can not be chosen as this value is not feasible. It does not mean that the repair rule can not introduce new conflict, as the newly selected value of the variable can violate another constraint.

I will give an explanation using a small example. Let us look again at the conflict matrix showing a constraint between variable v_2 and v_3:

$$\begin{array}{cc} & v_3 \\ v_2 & \left(\begin{array}{c|c|ccc} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ \hline 0 & \mathbf{1} & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{array}\right) \end{array}$$

If v_2 has value 3 and v_3 has value 2 ($v_2 = 3$ and $v_3 = 2$), a conflict occurs (in boldface). The repair rule now counts the number of zeros on row number 3 (number of non-conflicts of v_2) and on the column number 2 (number of non-conflicts of v_3) and repairs the variable with the smallest number of zeros this way the most restricted variable will be repaired. The new value of the variable is selected by taking a random choice among the feasible values of the variable. Feasible values are represented in the matrix by zeros, meaning non-conflict, therefore selecting a random zero-entry on the row or column that represents the variable is enough to find a value. In the example, both v_2 and v_3 have 3 zeros in their row and column, a variable is therefore selected randomly. Then a random number between 1 and 3 is taken. Suppose that variable v_2 is chosen and the random number is 2. The value of v_2 after the repair rule for this constraint would be 4, v_3 would stay unchanged. Then the next violating constraint in the individual would be repaired.

3.2.3 Constraint order

Another concern about the repair rule is in what order the different constraints will be repaired, *constraint order*. For instance, suppose I have a number of constraints that restrict the search space severely. If those constraints will be repaired early in the repair process, there is a high probability that the repair would be undone later in the repair process. The other way round is also possible. If the most restricting constraints would be repaired late in the repair process, they could restrict the search so severely early on, so that a solution will never be found, premature convergence. I propose a — total random — strategy which should insure that neither possibility can occur. The easiest way to do this is to take a random permutation of all constraints every time an individual is repaired. In this way each individual is repaired with a different random constraint order and therefore population diversity should be secured, negating both earlier stated problems.

3.3 Other considerations

The two sections above cover the two main issues about `ESP-GA`s. Other topics involved when using a `GA` will be discussed below. They involve the choices made on representation, crossover, mutation, selection and fitness.

The *representation* chosen is a set of integers. There will be n integers, each representing a different variable of the binary CSP. When a solution is found, the values of all integers break none of the constraints that were made by the constraint processing technique discussed earlier.

I have chosen the traditional one-point crossover for the *crossover operator* for ESP-GA. Although, in general, this crossover is considered to limit the search of the search space, I consider this positive because it limits the effect that the crossover operator has on the search process while it enhances the visibility of the effect of the repair rule. I have not examined what the effect on the results of ESP-GA is of having another crossover operator (like the uniform random crossover operator), on the results of ESP-GA.

I have used the *mutation operator* where the value of one variable is changed randomly per selected individual. Both variable and individual selection is done randomly. This mutation operator is chosen mainly to remain close to the original ESP-GA.

I have programmed ESP-GA as a pure steady state GA. This means that two selection methods can be distinguished, one selection method for selecting the parents of the genetic operators and one for selecting the content of the next generation. I have chosen the selection method for the parents of the genetic operators based on a *linear ranking scheme* while I chose the best individuals of the previous population, including the children of the genetic operators, as the selection method for the next generation. Although E. Marchiori has chosen roulette wheel selection for the original ESP-GA, I have not done so because I hope that by choosing these selection methods I hope to eliminate some of the random 'noise' that the original selection method introduces.

The *fitness function* I have chosen is the number of variables that violate a constraint. The *fitness maximum* is therefore n when all variables violate one or more constraints and a solution is found with a fitness value of zero, indicating that no constraints are violated. It is therefore obvious that a minimum has to be found. I suggest however, that — in later studies — the effect of using a different fitness-function is examined, namely the number of violated constraints. Although this fitness function gives the GA a much better insight in how far (off) it is in solving the CSP, I have not used it now because of the large number of constraints it has to check every time it is calculated when trying to solve problems with a high tightness and density. Although the worst case complexity of the first fitness function is the same as with the second fitness function, the probability that the first fitness function has to check all constraints seems to be smaller. Note however that, the further in the search, more and more constraints will be satisfied (as the GA comes closer to the solutions) and therefore the first fitness function will more and more revert to the worse case scenario. When a solution is found, all constraints have to be checked and the two fitness functions have the same complexity.

4 Solving CSPs with H-GAs

In [7], Eiben et al. investigate the possibility of using heuristics with GAs. Heuristics are commonly used in CSPs and are already available for most classical CSPs. Therefore, using them would be a natural solution to help solve CSPs when using GAs. In [7], two heuristic operators are specified: an asexual operator and a multi-parent operator. Both are used to maintain the constraints in CSPs and have been tested on two examples: *n-queens* and the *graph 4-coloring* problem. In this thesis I will call the GAs as suggested by Eiben et al. heuristic GAs: H-GAs.

The *basic idea* in [7] is to combine the essential probabilistic mechanisms in classical GAs with common heuristics used in traditional CSP solving methods. In H-GAs this is done by replacing the classical uniform random mechanism of the crossover and mutation operator by a crossover or mutation mechanism based on heuristics that take into account the prescribed constraints. The uniform random part of the GA and the new heuristic-based components are used to counterbalance each others deficiencies. The application of heuristics can improve the performance of the blind random mechanism while the random component can compensate the strong bias that is introduced by the heuristics.

As stated earlier, the use of traditional genetic operators have a negative effect on solving CSPs with GAs. Because they traditionally use randomness to alter individuals, they are blind to the constraints of the CSPs. This can result in altering an individual that satisfies some constraints in one population into an individual that violates those constraints. Instead of repairing an individual, as is done in ESP-GAs, in [7], Eiben et al. propose to change the genetic operators in a way that they maintain the constraints that an individual satisfies and change the variables that do not. This is done by two new genetic operators which include heuristics: an asexual operator that only changes one single individual and a multi-parent operator that introduces an individual based on two or more parents.

4.1 Asexual heuristic operator

The *asexual heuristic operator* selects a number of variables in a given individual, then selects new values for these variables. The amount of variables to be modified, the criteria for selecting these variable and the criteria for the new values of these variables are the *defining parameters* of the possible asexual operators. Different asexual operators can be denoted by the triple (n, p, g) where n indicates the number of variables to be modified — being either 1,2 or # (with # meaning that the number of variables to be altered is chosen randomly but is at most one-fourth of all variables in the individual) — and p and g indicate the selection criteria for variable and value selection respectively, denoted by an r for random selection and a b for a heuristic biased selection. In the H-GAs I study in this thesis, I will study an asexual operator that has the following defining parameters: $(\#, b, b)$, meaning that each time the operator is used, up to one fourth of the variables will be changed, its variables to be changed and the values they will be changed to are chosen using a heuristic.

4.1.1 Selecting the variable

In [6] some measures or *heuristics for variable selection* are mentioned:

1. number of constraints that are relevant to a specific variable, the cardinality of that variable;
2. number of unsatisfied constraints that are relevant to a specific variable, the unsatisfied cardinality of that variable[8];
3. average or minimum tightness of the constraints that are relevant to a specific variable;
4. number of possible values that a specific variable has;[9]
5. number of possible values that a specific variable has by arc to other variables; the number of arc-consistent values.

In [7], Eiben et al. choose to implement option 2, the asexual heuristic operator with a heuristic bias system that changes the variable that has the largest number of unsatisfied constraints that are relevant to the variable. It is expected that by changing this variable the largest improvement to the individual can be made.

4.1.2 Selecting the value

Also in [6] some measures or *heuristics for value selection* are mentioned:

1. the number of satisfied constraints that are relevant to the variable per value, the satisfied cardinality of the variable per value;
2. total number of possibilities for satisfying all relevant constraints;
3. the number of possibilities for satisfying the tightest constraint.

In [7], Eiben et al. choose to implement option 1, the value selection for the asexual heuristic operator with the measure that counts the number of satisfied constraints that are relevant to the variable, calculated per different value in the domain. It is expected that by using this measure on value selection, the possibility of introducing a (new) conflict in the individual is the smallest[10].

[8]This heuristic needs an instantiation to check if a constraint is satisfied or not.

[9]This heuristic is used for CSPs with a non-uniform domain size as otherwise it would simply be $|D_i|$.

[10]Note that this heuristic has to be calculated for every possible value the variable can have, thus m times

4.1.3 Implementation

There are three issues to the implementation of the asexual heuristic operator:

First issue is selecting the number of variables that will be chosen to be altered. This is implemented by taking a random number between one and $\lceil n/4 \rceil$ every time the asexual heuristic operator is used. Variable selection is repeated that many times.

The second issue is the implementation of variable selection in such a way that it works best with the output generated by the `CSP` instance generator. I have found that the easiest way to do this is by calculating a constraint matrix. With the constraint matrix it is possible to find simply and quickly all constraints relevant to a variable. The heuristic has to check for every variable in the individual how many of the relevant constraints to the variable are currently satisfied by the values of the variables in the individual. This is done by checking for every relevant constraint, if, for the variables of the constraint, the values of the individual result in a conflict. This is done by checking if the conflict matrix of the constraint has a non-zero value on the entry specified by the values of the variables. The variable with the most unsatisfied constraints found in this way, is chosen to be changed.

An example may explain more clearly how this measure works and how I have implemented it. As said earlier, the easiest way to find the set of relevant constraints to a variable is by looking in the constraint matrix. Using the ongoing example and the output as it is given in appendix A, the following constraint matrix can be calculated:

	v_1	v_2	v_3	v_4	v_5
$c_{2,3}$	0	1	1	0	0
$c_{2,5}$	0	1	0	0	1
$c_{3,4}$	0	0	1	1	0
$c_{3,5}$	0	0	1	0	1
$c_{4,5}$	0	0	0	1	1

For individual $\langle p_1, p_2, p_3, p_4, p_5 \rangle = \langle 1, 2, 3, 4, 5 \rangle$, in the conflict matrix of $c_{3,4}$ and $c_{4,5}$ on entries $(3,4)$ and $(4,5)$ a non-zero entry was found ($C_{3,4} = 1$ and $C_{4,5} = 1$). All other constraints have a zero-entry in the conflict matrix for their respective variables ($c_{2,3}(2,3) = 1$, $c_{2,5}(2,5) = 1$ and $c_{3,5}(3,5) = 1$). This means that only $c_{3,4}$ and $c_{4,5}$ are unsatisfied. Knowing this, it is clear that v_1 and v_2 have no unsatisfied relevant constraints, v_3 and v_5 have one unsatisfied relevant constraint while v_4 has two unsatisfied relevant constraints. Therefore the measure will select v_4 as candidate for a change of its value.

The third and final issue is the implementation of the value selection measure so that it works best with the output generated by the `CSP` instance generator. The value selection heuristic counts for every constraint relevant to the variable and the whole domain minus its current value, how many of relevant constraints are satisfied. Using the constraint matrix again is a simple way to calculate this heuristic.

Using the ongoing example, the number of satisfied relevant constraints of

v_4, that was selected with the previous measure, will be calculated. From the constraint matrix it is clear that constraints $c_{3,4}$ and $c_{4,5}$ are relevant to variable four. The conflict matrix of these combinations of variables are:

$$\begin{array}{cc} & v_4 \\ v_3 & \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & 1 & \mathbf{0} \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

and

$$\begin{array}{cc} & v_5 \\ v_4 & \begin{pmatrix} 1 & 0 & 1 & 1 & \mathbf{1} \\ 0 & 1 & 1 & 0 & \mathbf{0} \\ 1 & 0 & 0 & 0 & \mathbf{0} \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & \mathbf{1} \end{pmatrix} \end{array}$$

Again for individual $\langle p_1, p_2, p_3, p_4, p_5 \rangle = \langle 1, 2, 3, 4, 5 \rangle$ it is now clear that I have to count the number of conflicts of variable v_4 that it has with variables three and five. The row to be checked with constraint $c_{3,4}$ and the column to be checked with constraint $c_{4,5}$ are boldfaced. As the value of 3 for v_4 has no conflicts with either of the other variables, it will be the new value for v_4. All other values have at least one conflict with one of the constraints[11].

4.2 Multi-parent heuristic operator

The *multi-parent heuristic operator* of H-GAs is a multi-parent operator that uses a heuristic to determine which value of its parents is selected for its child. The basic mechanism of multi-parent operators is scanning [9]. This means that the operator examines all variables of the parents consecutively and per variable chooses one of them for the child. By using special marker update mechanisms, e.g. shifting the markers to the first value that does not yet occur in the child, the scanning technique can be adjusted to special types of problems like permutation based problems, such as routing or scheduling. Scanning can also be enriched by problem dependent heuristics relying on extra information, e.g. edge length for routing problems. There are several ways to choose between the parent variables[12]:

- uniform random; choosing the value at random, no heuristic is used;
- occurrence base; the value that occurs the most is represented in the child);
- biased on the fitness of the parent;

[11] Value 4 of variable v_4 is not boldfaced and is not checked. This to ensure that a new value for the variable will be chosen.

[12] Note that some of the basic choices that were also present in value selection are also present here: a random strategy, and using problem independent or problem dependent heuristics

- using a problem dependent heuristic.

The *implementation of the multi-parent heuristic operator* is done in a way that any number of parents, from two up, can be used. For the value selection method I will use a problem dependent heuristic. Earlier I've given alternatives for value selection and the same one I have used with the asexual heuristic operator I will use here as well: the number of satisfied constraints that are relevant to the variable. The difference with the asexual heuristic operator is that the heuristic will not evaluate all possible values but only the values that are represented by the parents. Note that Eiben et al. uses the same heuristic in the original H-GA in [7]. An example of the value selection heuristic is already given earlier, I will not repeat it here.

4.3 Other considerations

The *representation* used for the individuals in this version of the H-GA is a string of integers each representing a different variable. A solution is therefore a individual which values violate none of the constraints. This representation of an individual is identical to that of ESP-GA. The difference between H-GAs and traditional GAs is of course the use of the two new heuristic operators that replace the original crossover operator and/or the mutation operator. If only one of the original operators is replaced however, the other still remains. The *original crossover operator* is the traditional one-point crossover while the *original mutation operator* changes a random variable to a random value in its domain. *Fitness evaluation* is done by counting the number of variables that violate any of their relevant constraints. The same remark as was made with regard to the fitness function in ESP-GAs can be made here also: counting the number of violated constraints might give the GA a better insight in how far (off) it is to a solution. Again I have not done so now because of the time-consuming nature of the fitness function. As was done in the ESP-GA I have used a *linear ranking scheme* as the selection-method.

The heuristics used in the two heuristic operators can be quite expensive to use. This can be changed in a positive way by *precalculating the conflict matrices* so that looking up a single bit in five different arrays can be exchanged for looking up a single byte in a single array and then using bitshifts to gain the specific bit. This means doing a lot of work (mostly bitshifts to compress the conflict matrix) at the start of the solving process but doing significantly less calculation during the solving process, while at the same time, reducing the amount of storage needed. I have not used this method mostly for simplicity reasons, thereby sacrificing mainly processortime. It may be interesting however to study the improvement on the calculating time in applications where this issue is more important.

5 Solving CSPs with Arc-GAs

In [31, 32] M.C. Riff-Rojas describes her alteration of the classic GA for solving CSPs. Noting that until now little development in the field of solving CSPs with GAs using the constraint network was done, she describes a GA that solves CSPs by altering the fitness function [31] and the genetic crossover and mutation operators [32]. The fitness function, called *arc-fitness function*, was altered in such a way that individuals, whose values have more arcs that satisfy their constraints, have a preference over other individuals. To find the sets of arcs to be examined, the arc-fitness function uses information derived from the constraint network. The genetic operators, called the *arc-crossover operator* and the *arc-mutation operator*, also use information about the constraint network to generate offspring that has the best combination of the values of the parent variables. This is done by defining two partial fitness functions for both operators.

The basic idea of the *arc-fitness function* is to give preference to individuals that have more arcs in the constraint network which do not violate their constraint. This is done by evaluating all constraints in the CSP. Every constraint is checked if the values of the individuals satisfy it or not. If the constraint is satisfied, the so called error evaluation of the constraint is zero. If not, the number of relevant variables[13] and the variables that are connected by arc to these variables, are counted.

The set of these variables I call the *error evaluation set of variables* of the constraint and it is found by examining the constraint network. The number of variables in this set is independent of the individual, it is however dependent on the instantiation of the individual if the set is calculated over a partial instantiation, as constraints whose variables are not instantiated — or not completely instantiated — are not taken into account. The cardinality of the error evaluation set, or the number of variables in the set, is called the *error evaluation* of the constraint. The sum of the error evaluations of all violated constraints — the cardinality of the error evaluation sets of all constraints — is the *arc-fitness value* of the individual.

A picture is useful when explaining which variables are added to the error evaluation set of variables. In picture 1 nodes that are filled-in are instantiated variables while nodes that are not filled-in are not instantiated. Suppose I am calculating the error evaluation set of variables of constraint C_1, we can see that its variables v_1 and v_2 are connected with variables v_3, v_4, v_5, v_6 and v_7. If one of the two, or both of the variables will change, constraints C_2, C_3, C_4, C_5 and C_6, may become unsatisfied, while they could be satisfied now. The resulting error evaluation set of variables is: $\{v_3, v_4, v_5, v_6, v_7\}$ [14].

If the error evaluation set of variables was calculated over a partial instantiation, as can happen in the arc-crossover operator, the resulting error evaluation set of variables would be: $\{v_3, v_5, v_7\}$, as only the related variables of completely

[13]in a binary CSP there are two relevant variables to every constraint

[14]When calculating the error-evaluation set of variables for fitness evaluation, all variables are instantiated. The instantiation in figure 1 is added for use with error-evaluation over a partial instantiation.

instantiated constraints are counted.

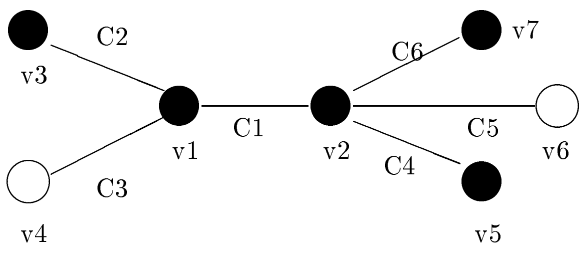

Figure 1: Explanatory picture of the error evaluation set of variables.

The *basic idea of the arc-crossover operator* is to generate a new child from two randomly selected parents. The child inherits the best combination by arc of the parent's values. The arc-crossover operator makes an iterative procedure over the constraints by arc. The constraints are ordered based on the error-evaluation of the constraint, thereby making sure that the first arc to analyze is the heuristically hardest constraint to satisfy in the constraint network because it has the highest value in the error evaluation function.

The *basic idea of the mutation operator* is to generate a new individual by altering a randomly chosen variable in the individual to a new value. This value is selected by calculating for every constraint — that is relevant to the variable selected — its error evaluation value for every possible value that the selected variable can have.

As stated earlier, the arc-fitness function, the arc-crossover operator and the arc-mutation operator all have their own (partial) fitness functions. The fitness function is already explained earlier which leaves the (partial) crossover fitness function and the (partial) mutation fitness function. Both have sets of constraints, that the fitness functions are calculated over. Both sets of constraints as the corresponding fitness functions will be discussed below, all are also defined mathematically by M. Riff-Rojas in [32].

The set of constraints needed for the partial arc-crossover fitness function, called the *set of crossover instantiated constraints*, is assembled started by a constraint and then checked to see if all relevant variables are instantiated. If this is the case, add the constraint to the set. Then every constraint that shares a variable with this constraint is also checked to see if its variables are all instantiated. If this is the case, the constraint is also added to the set[15]. In short, one could say that the set of crossover instantiated constraints are all relevant instantiated constraints to the two variables of the starting constraint.

Again, a picture may explain the set of crossover instantiated constraints better. I have copied the picture that I used for explaining the error evaluation set of constraints in picture 2, because the set of crossover instantiated

[15]The set is assembled with a constraint as its start because the arc-crossover operator uses a iterative procedure by constraint.

constraints also starts with a constraint. With the set of crossover instantiated constraints, there is always an instantiation. Again, filled-in nodes are instantiated while those that are not, are also not instantiated. Only constraints that have all variables instantiated are added to the set. Suppose I want to calculate the set of crossover instantiated constraints of constraint C_1, we find the set: $\{C_1, C_2, C_4, C_6\}$.

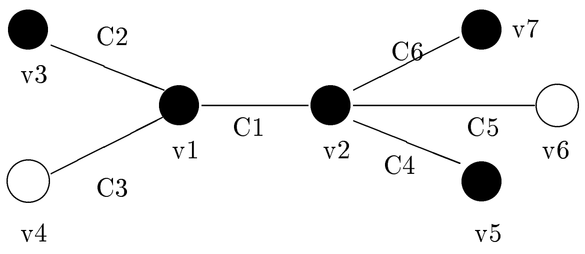

Figure 2: Explanatory picture of the set of crossover instantiated constraints

The *partial crossover fitness function* (cff) of a specific constraint is the sum of all error evaluation values of the constraints in the set of crossover instantiated constraints over that constraint.

The set of constraints needed for the partial arc-mutation fitness function, called the *set of mutation instantiated constraints*, is assembled started by a variable. All relevant constraints to the variable are checked if they are instantiated, and if so added to the set[16]. Again, in short, one could say that the set of mutation instantiated constraints are all relevant instantiated constraints to the starting variable.

Again, I will use a picture to explain how the set of mutation instantiated constraints is calculated. Picture 3 is unlike the first two pictures because the set of mutation instantiated constraints is calculated with a variable as starting point. Like the two earlier pictures, the instantiated variables are pictured like filled-in nodes while uninstantiated variables are pictured like nodes that are not filled-in. Suppose I want to calculate the set of mutation instantiated constraints of variable v_1, then only the completely instantiated constraints C_1, C_2 and C_4 are added to the set. The resulting set of mutation instantiated constraints is then: $\{C_1, C_2, C_4\}$.

The *partial mutation fitness function* (mff) of a specific variable is the sum of all error evaluation values of the constraints in the set of mutation instantiated constraints over the given variable.

It is important to note that different values for the variables in both the arc-fitness function, the partial crossover fitness function and the partial mutation fitness function produce different error evaluation function values because they change the number of variables in the error evaluation set of constraints since

[16]Here the set is assembled with a variable as starting point because the arc-mutation operator operates on variables

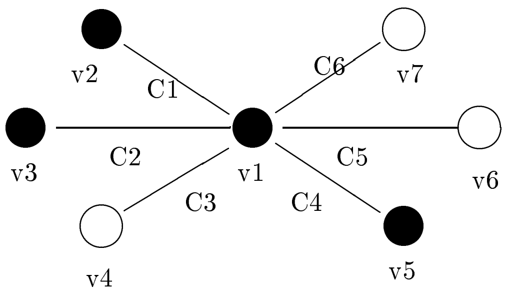

Figure 3: Explanatory picture of the set of mutation instantiated constraints

they can satisfy other constraints. Different instantiations result of course also in different results for the partial fitness functions.

After these definitions the *algorithm for the arc-crossover operator* can be described as follows:

First two parents are selected randomly. At first, the child to be made from these parents has no variables instantiated. With the evaluation of the first constraint in the ordered set of constraints, two variables will be instantiated. Later in the process, some variables will already be instantiated, the other variable to be instantiated will be chosen between the two parents based on the lowest value for the partial mutation fitness function (mff) of the variables. If none of the variables are instantiated there are three possible choices:

- the constraint is satisfied in both parents: assign the values of the constraint from the parent with the best fitness evaluation;
- the constraint is satisfied for one parent: assign the values of the constraint from this parent;
- the constraint is not satisfied in both parents: assign the best combination of values of variables from both parents based on the partial crossover fitness function (cff).

The *algorithm of the arc-mutation operator* can be described by randomly selecting a variable to mutate and then selecting the value of the variable to be changed by applying the partial mutation fitness function to all values of the domain except the present value. The value with the smallest value for the partial mutation fitness function will be chosen.

5.1 The Arc-fitness function

In the *implementation of the fitness function* as well as the crossover and mutation fitness functions, it is convenient to have a constraint matrix. With the constraint matrix, constructing the error evaluation set of constraints is simple. Finding the variables that are relevant to the constraint under investigation is done by searching the two non-zero entries in the constraint matrix on the row for the appropriate constraint. Finding the constraints that share a variable with the constraint under investigation is done by finding the non-zero entries in the columns of the two relevant variables of the constraint. The constraints that are relevant to the thus found variables are the variables that share a variable with the constraint under investigation. The error evaluation set of variables is then found by adding to it all variables of the thus found constraints.

The fitness value of a constraint is found by only counting those variables in the error evaluation set of variables that belong to a constraints whose values in the individual under investigation violate the constraint to whose arc the variables belong. If all constraints in the `CSP` are checked in this way, the resulting error evaluation value over all these constraints is the fitness value of the individual.

Once more I will use the ongoing example to clarify the procedure. The constraint matrix has already been given in the section about `H-GA`s:

	v_1	v_2	v_3	v_4	v_5
$c_{2,3}$	0	1	1	0	0
$c_{2,5}$	0	1	0	0	1
$c_{3,4}$	0	0	1	1	0
$c_{3,5}$	0	0	1	0	1
$c_{4,5}$	0	0	0	1	1

When investigating individual $\langle p_1, p_2, p_3, p_4, p_5 \rangle = \langle 1, 2, 3, 4, 5 \rangle$ it is found that constraints $c_{3,4}$ and $c_{4,5}$ are violated. Let's start with constraint $c_{3,4}$, its two relevant variables are v_3 and v_4. Constraint $c_{3,4}$ shares v_3 with constraints $c_{2,3}$ and $c_{3,5}$, therefore changing variable v_3 in any way will have effect on these two constraints as well. Constraint $c_{3,4}$ shares v_4 with constraint $c_{4,5}$ only. To calculate the set of error evaluation of constraint $c_{3,4}$ is done by naming the variables of these four constraints ($c_{3,4}, c_{2,3}, c_{3,5}$ and $c_{4,5}$) and is: $\{v_2, v_3, v_4, v_5\}$. The error evaluation of constraint $c_{3,4}$ is the cardinality of this set and is therefore 4. The same can be done to constraint $c_{4,5}$. It's 'sharing' constraints are $c_{3,4}$, $c_{3,5}$ and $c_{2,5}$ and its set of error evaluation variables is: $\{v_2, v_3, v_4, v_5\}$, and its error evaluation is 4. The error evaluation of the given individual is calculated by adding all error evaluations of all unsatisfied constraints of the individual: $4 + 4 = 8$.

5.2 Arc-mutation and Arc-crossover

When calculating the error evaluation set of constraints and thus the error evaluation of a constraint I use the complete instantiation while in arc-crossover and arc-mutation, the partial instantiation is used. This can be calculated by not including a number of variables in the constraint matrix. This is implemented by keeping a list of variables that are instantiated. Before adding a variable to any set (this includes the crossover and mutation sets) this list is checked. If the variable is not on the list, it will of course not be added to the sets. The same can be said about a constraint with one of its variables not in the partial instantiation, it also will not be added to the set.

Implementing the set of crossover instantiated constraints is very similar to calculating the error evaluation set of a constraint. Again, I begin by looking on the row of a constraint and adding the variables of the constraint if they are instantiated. After this I check what constraints share a variable by adding to a list, the constraints that have a non-zero entry in the column of the first two variables. After this I add the original constraint to the list and the crossover instantiated constraints list is complete. The main difference between the error evaluation set of a constraint and the crossover instantiated constraints is that the latter is a set of constraints while the first is a set of variables. The method of extracting the information however is nearly the same.

Calculating the partial crossover fitness value of an individual is done by calculating the fitness value of the individual but only evaluating the constraints in the set of crossover instantiated constraints. This means that the fitness function as explained earlier is used only for checking the instantiated variables.

Again, let's use an example to clarify what I have just explained. With the partial crossover fitness function, I too need the constraint matrix:

$$\begin{array}{c|ccccc}
 & v_1 & v_2 & v_3 & v_4 & v_5 \\
\hline
c_{2,3} & 0 & 1 & 1 & 0 & 0 \\
c_{2,5} & 0 & 1 & 0 & 0 & 1 \\
c_{3,4} & 0 & 0 & 1 & 1 & 0 \\
c_{3,5} & 0 & 0 & 1 & 0 & 1 \\
c_{4,5} & 0 & 0 & 0 & 1 & 1
\end{array}$$

The largest difference with the arc-fitness function is however, that the crossover fitness function is calculated over a partial instantiation. In this example I use the partial instantiation: (v_3, v_4, v_5). Suppose I want to calculate the partial crossover fitness over constraint $c_{3,5}$. Both its variables are instantiated so the constraint is added to the set of crossover instantiated constraints. The constraints that share variables v_3 with constraint $c_{3,5}$ are $c_{3,4}$ and $c_{2,3}$ but only $c_{3,4}$ is completely instantiated and therefore added to the set of crossover instantiated constraints. The constraints that share variable v_5 with constraint $c_{3,5}$ are $c_{2,5}$ and $c_{4,5}$ but this time only $c_{4,5}$ is completely instantiated and therefore added to the set. The set of crossover instantiated constraints is now: $(c_{3,5}, c_{3,4}, c_{4,5})$. The crossover fitness function is now calculated by using the same method as with the arc-fitness function but now only calculating the error evaluation over the constraints in the set of crossover instantiated constraints. As this is explained earlier, I will not repeat it here.

Implementing the mutation instantiated constraints is again very similar to calculating the crossover instantiated constraints in that it involves a set of constraints and uses the same two basic search actions along the rows and columns of the constraint matrix. The difference however lies in the fact that the search does not start with a specific constraint but with a specific variable. The search starts by checking the constraints that are relevant to the variable.

Then all constraints are checked to see if they are completely instantiated, by checking if the other variable of the constraint is instantiated. All completely instantiated constraints are added to the set.

As it was the case in the partial crossover fitness function, the partial mutation fitness value of an individual is calculated by calculating the fitness value of the individual but only evaluating the constraints in the set of mutation instantiated constraints. Again only a partial instantiated fitness function is used.

Here also I will use a small example to clarify the method. I will not change the preliminaries from last example. The instantiation is still: (v_3, v_4, v_5), the individual is still: $\langle 1, 2, 3, 4, 5 \rangle$ and the constraint matrix is of course still:

$$\begin{array}{r|ccccc}
 & v_1 & v_2 & v_3 & v_4 & v_5 \\
\hline
c_{2,3} & 0 & 1 & 1 & 0 & 0 \\
c_{2,5} & 0 & 1 & 0 & 0 & 1 \\
c_{3,4} & 0 & 0 & 1 & 1 & 0 \\
c_{3,5} & 0 & 0 & 1 & 0 & 1 \\
c_{4,5} & 0 & 0 & 0 & 1 & 1
\end{array}$$

Suppose I want to calculate the partial mutation fitness value for variable three (v_3). Constraints $c_{2,3}$, $c_{3,4}$ and $c_{3,5}$ have this variable. Constraint $c_{2,3}$ however is not completely instantiated and will therefore not be included into the set of mutation instantiated constraints. This set is now: $\{c_{3,4}, c_{3,5}\}$. The partial mutation fitness value of variable v_3 is now calculated by taking the arc-fitness value over the individual while only checking the constraints in the set of mutation instantiated constraints.

With Arc-mutation I randomly select a individual and randomly select the variable that I want to mutate. Then I calculate the partial mutation fitness value for every value in the domain of that variable excluding the current value. The new value for the variable will be the value with the lowest partial mutation fitness value.

Before Arc-crossover can be done, all constraints have to be ordered after their error evaluation value, highest value first. Then I randomly select two parents and start the crossover process. At first no variables are instantiated but every time a constraint is checked, new variables will be instantiated. The crossover process is begun by checking the first constraint in the ordering. The first check that is made is checking how many parents are yet instantiated. If only one parent is not instantiated yet, the child variable is chosen among the parents by calculating the mutation fitness function of the parents and choosing the variable with the lowest value. If both parents are not instantiated yet, three options can occur based on how the constraint is satisfied in the parent individuals. If the constraint is satisfied in both parents, the two variables are chosen from the parent with the best fitness value. If the constraint is satisfied in only one of the parent individuals, the value of the variables to be instantiated are copied from this parent. If the constraint is not satisfied by any of the parent, the values are combined with each other (there are now two possible combinations) and they are evaluated by the crossover fitness function. Again the combination with the lowest value is copied into the child individual.

5.3 Other considerations

M.C. Riff Rojas, altered the traditional GA in three ways. She replaced the original fitness function by an arc-fitness function which gives preference to individuals that satisfy more arcs in the constraint network. Next to that, she replaces both the traditional crossover and mutation operators with counterparts that look to the constraint network for better evaluation. One thing that remained the same was the integer representation of the individuals. Again I have used the same *representation* that was used in ESP-GA, an individual is a string of integers that represent each a variable in the CSP. Crossover and mutation rate will be discussed in the section about the experimental setup.

6 Software description and experimental setup

In the following three subsections, the specific software description and experimental setup of all three methods is given. However, in general, some things about the software description and experimental setup can be said.

An effort was made to incorporate the specific details of the three methods into a GA that is well known. In this way the differences between the three methods can be pointed out better because all other parts of the GA have well known characteristics. The GA I have chosen was a pure steady state GA with the classic one point crossover and total random mutation. The selection method for the parents of the genetic operators uses linear ranking strategy while the selection method of the next generation is to select the best individuals from the intermediate population.

A pure steady state GA produces per generation from a number of parents an equal number of children with the crossover operator which in turn all get mutated by the mutation operator. These children are then added to the current population where they compete with the rest of the population, including their parents, to get in the next generation, elitist selection.

One point crossover takes two parents. Then it selects a single crossover point. The first child is constructed by taking the first part up to the crossover point from the first parent and taking the rest from the second parent. The other child is made by reversing the process, taking the first part from the second parent and the rest from the first parent.

The total random mutation operator produces a single child from a single parent by randomly taking a single variable in the individual and assigning a random value from its domain to it.

In this section I use small tables to systematically explain what parts in the general GA are used and which parts are exchanged by parts from the different methods. Table 2 shows the short overview for the general GA.

Crossover operator	One-point crossover
Mutation operator	Total random mutation
Fitness function	Number of violated constraints
Parent Selection method	Linear ranking selection
Selection method	Elitist selection

Table 2: Short overview of the experimental setup of the basic-GA

I have used the GA-library *Leap* for producing the three GAs. Using this library, that was developed by J. van Hemert, meant that I only had to implement the parts of the GAs that were different from the GAs already in the library. The library was linked with the random CSP instance generator, which was also developed by J. van Hemert. This made checking if there was a constraint or a conflict in the CSP fairly easy. In the GA-library, the above mentioned 'standard' GA was already implemented and that implementation was used.

The general setup of the experiment was to have population size of 10 and perform per generation a single crossover operation and two mutation operations, resulting in three fitness evaluations per generation.

6.1 Software description and experimental setup ESP-GA

As was explained in section 3, ESP-GA is an adjustment of a GA because it adds to the GA a repair algorithm that repairs all individuals just before the selection of the next population. None of the operators or selection methods are changed by this method therefore the table of this methods is very similar to the table of the basic GA, it only adds an entry to the extra line. The repair algorithm is activated just after reproduction, but before the new population is chosen. By repairing the newly created individuals I hope that the exploration by the genetic operators is enhanced by repairing the new individuals.

Crossover operator	One-point crossover
Mutation operator	Total random mutation
Fitness function	Number of violated constraints
Parent Selection method	Linear ranking selection
Selection method	Elitist selection
Extra	Repair rule

Table 3: Short overview of the experimental setup of ESP-GA

6.2 Software description and experimental setup H-GA

In [7], Eiben et. al do not describe a GA as such, but describe two new operators, the earlier explained heuristic asexual and heuristic multiparent operators. Noting that the heuristic asexual operator can also be used as a mutation operator

— as it changes a single individual — I see three possibilities for a GA that I have investigated:

1. Use the heuristic asexual operator as a replacement of the crossover operator in the traditional GA.
2. Use the heuristic multiparent operator as a replacement of the crossover operator in the traditional GA.
3. Use the heuristic multiparent operator as a replacement of the crossover operator and the heuristic asexual operator as a replacement of the mutation operator in the traditional GA.

All three versions were implemented and compared to the other two methods.

	Version 1	Version 2	Version 3
Crossover operator	Asexual heuristic operator	Multi parent operator	Multi parent operator
Mutation operator	Total random mutation	Total random mutation	Asexual heuristic operator
Fitness function	Number of violated constraints		
Parent Selection method	Linear ranking selection		
Selection method	Elitist selection		
Extra	None		

Table 4: Short overview of the experimental setup of the three versions of H-GA

6.3 Software description and experimental setup Arc-GA

Arc-GA is an extensive overhaul of the general GA. It changes both the crossover and the mutation operator and uses a new fitness function to evaluate the population. Although in [31] a new selection method is also proposed, I have not implemented it. I have done this for two reasons. One reason is that this method was not used in [32], the other reason is that the new selection method was not designed for steady state GAs. The new selection method was not mentioned in [32] because most of the additional abilities of the this selection method where incorporated in the fitness function in [32]. By using this fitness function I also have the benefit of the abilities of the selection method. In table 5, a short overview of Arc-GA is given.

7 Results

The results in table 6 were found by running the three methods (five if you count the three versions of the H-GA), on 10 instances of the 25 combinations of

Crossover operator	Arc-crossover operator
Mutation operator	Arc-mutation operator
Fitness function	Arc-fitness
Parent Selection method	Linear ranking selection
Selection method	Elitist selection
Extra	None

Table 5: Short overview of the experimental setup of `Arc-GA`

constraints tightness and density. On each instance, 10 different runs were made. In total there were 2500 runs for each method, 100 runs for every combination of constraint tightness and density. The large number of runs should ensure that the values in the table are a correct representation of the performance of the methods.

The `CSP` instances were `CSP`-problems of 15 variables, all having a domain size of 15. Although much larger `CSP` problems can be created and hopefully solved, using the three methods, this is a fairly general problem size which can be calculated without burdening the comparison with exceptionally long calculation times. In order to try to find a solution even when trying to solve hard `CSP`s, I have put the maximum number of operations on 100000. This means that the `GA` stops the search after having done 100000 operations on the population.

For the comparison I used two common measures. First, the percentage of problems solved, the *success rate* (SR) of the `GA`. If one method can still solve a `CSP` while the other can not, the first is generally considered the better one. The same holds for different percentages, if one method solves the `CSP` most of the time while another method only solves it rarely, the first method is considered better. The second measure is the *average* number of *evaluations to solution* (AES). If one method can calculate a solution of a `CSP` in less evaluations than another method it is also considered better[17].

Notice that in density-tightness combination (0.1, 0.1), for several methods, instead of the number of evaluations to solution, the symbol i has been placed. This is done to indicate that a solution was found in the initial population. The number of evaluations to solution of these values is the size of the population as, in my implementation, before checking if there is already a solution, all individuals of the initial population are evaluated. Some may argue that this process should be interrupted immediately after finding a solution. This would however unbalance the results of these methods if they are compared to the methods that need a few more evaluations to find a solution. I believe that this alteration in the result table removes this unbalance.

Although the time used by the method is of some interest, I have not used it in my comparison. The reason for excluding user-time from the comparison is that there can be numerous factors that influence the time that is needed

[17]if a run does not find a solution to the `CSP`, AES is undefined and not included into result. In such a run, no evaluations are counted for calculating AES.

den-sity	alg	tightness				
		0.1	0.3	0.5	0.7	0.9
	`ESP-GA`	**1(*i*)**	**1(23)**	1(78)	0.91(600)	0.45(13559)
	`H-GA.1`	1(11)	1(54)	1(169)	1(643)	**0.72(10419)**
0.1	`H-GA.2`	1(12)	1(88)	1(315)	1(1325)	0.61(15254)
	`H-GA.3`	**1(*i*)**	**1(23)**	**1(53)**	**1**(484)	0.64(14752)
	`Arc-GA`	**1(*i*)**	1(32)	1(79)	0.99**(211)**	0.27(14131)
	`ESP-GA`	**1(23)**	1(132)	0.91(5699)	0.01**(8366)**	0()
	`H-GA.1`	1(50)	1(441)	**1**(4481)	0.02(69632)	0()
0.3	`H-GA.2`	1(70)	1(704)	1(4921)	**0.05**(22954)	0()
	`H-GA.3`	1(26)	**1(119)**	0.97(3587)	0()	0()
	`Arc-GA`	1(33)	1(175)	0.91**(617)**	0.02(25802)	0()
	`ESP-GA`	**1(36)**	1(891)	**0.19**(4371)	0()	0()
	`H-GA.1`	1(121)	1(1671)	0.08(43337)	0()	0()
0.5	`H-GA.2`	1(188)	1(1861)	0.07(36780)	0()	0()
	`H-GA.3`	1(47)	1(498)	0.07(21083)	0()	0()
	`Arc-GA`	1(95)	**1(388)**	0.01**(554)**	0()	0()
	`ESP-GA`	**1(52)**	0.91(8190)	0()	0()	0()
	`H-GA.1`	1(204)	**1**(5950)	0()	0()	0()
0.7	`H-GA.2`	1(428)	1(8454)	0()	0()	0()
	`H-GA.3`	1(61)	0.95(8960)	0()	0()	0()
	`Arc-GA`	1(138)	0.71**(1230)**	0()	0()	0()
	`ESP-GA`	**1(69)**	**0.42**(12180)	0()	0()	0()
	`H-GA.1`	1(338)	0.37(35593)	0()	0()	0()
0.9	`H-GA.2`	1(487)	0.4(32954)	0()	0()	0()
	`H-GA.3`	1(92)	0.13(21457)	0()	0()	0()
	`Arc-GA`	1(164)	0.04**(1193)**	0()	0()	0()

Table 6: Success rates and the corresponding average number of evaluations to solution (within parenthesis) for `ESP-GA`, `Arc-GA` and the three versions of `H-GA`

to calculate a solution to a `CSP`, not the least of which is the specific computer system that was used. A powerful computer can calculate a solution much faster than a slow computer. Using user-time for this comparison would probably say more about the performance of the used computer systems then it would about the used methods. Furthermore, if user-times were used, they would probably be out-of-date in a short while as the performance of computer systems has been dramatically increased in the recent years.

8 Comparison

Comparisons between the three methods are made based upon the results they had in the earlier explained tests. These results are put down in result table 6. As explained earlier, the first value in the result table indicates the success rate of the method while the second value (within parenthesis) indicates the corresponding average number of evaluations to solution. I have emphasized the best results in the table. In cases where all three methods found a solution in all there runs, a SR of one, I have boldfaced the value of the methods with the least average number of solution. In cases where not all methods had a SR of one, I boldfaced the best SR separate from the best AES. This resulted in boldfacing a SR of one for one method while boldfacing the least number of AES in another method for the same density-tightness combination. This seems somewhat awkward in density-density combinations (0.3,0.7), (0.5,0.5), (0.7,0.3) and (0.9,0.3) and to a lesser extend (0.1,0.7). I have done this to show that a method that has the best SR does not necessarily has the least AES.

The results from the result table do not tell the whole story about the performance of the GA. Although SR and AES tell a lot about the performance, especially when using heuristics to help GAs to solve CSPs, there is always work done by the GA that is not measured by these two measures. The amount of work that is done by the GA can be deducted by the amount of (user) time that the GA needs to solve the CSP. As I have already explained, I have not used user time as a measure but I have used it to make some suppositions about the *hidden work* a GA does. The three versions of H-GA seem to do the least amount of hidden work comparable to the other measures while ESP-GA seems seems to do the most amount of hidden work with Arc-GA somewhere between these two GAs.

This can partly explained by examining the algorithms. H-GA was designed to include heuristics directly in the heuristic operators. This means that the heuristics are only calculated when they are needed. This introduces the least amount of hidden work. And although Arc-GA also uses heuristics, these are more complex than those used in H-GA. For every used heuristic a specific set of variables or constraints has to be calculated. Furthermore, a special table has to be compiled and maintained throughout the solving process. Arc-GA also uses heuristics more often, in the crossover and mutation operator as well as in the fitness function. ESP-GA introduces the largest amount of hidden work. Although the used heuristics are not as difficult to calculate as those used in Arc-GA, ESP-GA does check every constraint and repairs it when it is violated. In CSPs that are large and complex this means that a lot of constraints have to be evaluated, even when these constraints are not violated. In addition, ESP-GA uses and maintains the γ-table, a special table with all constraints and their conflicts, which also adds hidden work.

8.1 About ESP-GA

The results in table 6 show that ESP-GA performed best of all three GAs in density-tightness combinations (0.1,0.1), (0.1,0.3), (0.3,0.1), (0.5,0.1), (0.9,0.1). In the first combination, (0.1,0.1), it shares this performance with Arc-GA and the third version of H-GA while it shares the performance in the second combination with the third version of H-GA. ESP-GA also has the best performance with regard to SR in density-tightness combinations (0.5,0.5) and (0.9,0.3) and it performs best with regard to AES in density-tightness combination (0.3,0.7).

In general, I noticed that ESP-GA had a lesser performance when looking at

SR than the first and second version of H-GA. This is seen clearly in density-tightness combinations (0.1,0.7), (0.1,0.9), (0.3,0.5), (0.3,0.7) and (0.7,0.3) while ESP-GA has a better success rate than one of the two first versions of H-GA in density-tightness combinations (0.5,0.5) and (0.9,0.3). On the other hand ESP-GA has a better AES in most of the density-tightness combinations except for combinations (0.1,0.9) and (0.7,0.3) where the first version of H-GA has a better AES and combination (0.3,0.5) where both the two first versions of H-GA have a better AES.

8.2 About the three versions of H-GA

According to the results in table 6, H-GA performed best in density-tightness combinations (0.1,0.9) for version one and (0.1,0.1), (0.1,0.3), (0.1,0.5) and (0.3,0.3) for version three. For the combinations (0.1,0.1) and (0.1,0.3) for version three, one must note that the first combination has a shared best performance with ESP-GA and Arc-GA while the second combination has a shared best performance with ESP-GA. With regard to the performance in SR, version one of H-GA performed best in density-tightness combinations (0.3,0.5) and (0.7,0.3) while version three performed best in combination (0.1,0.7).

From the results of all three versions of H-GA, I can say that the first two versions of H-GA perform slightly better than the third version. When CSPs get harder to solve, it is the third version that sometimes does not find a solution. It can also be said however that the third version, in general, uses the least AES. Only in density-tightness combinations (0.1,0.9) and (0.7,0.3) do the other GAs, on average, use less evaluations. In density-tightness combination (0.3,0.7), the third version can not find a solution to the CSP. It must be said however, that all other GAs find only a few solutions here also, this means that the failure to find a solution here can not be used to conclusively say that the third version performs the worst of all GAs.

The difference between the first and the second version of H-GA is much less obvious. The first version has a somewhat better SR than the second version. Only in density-tightness combinations (0.3,0.7) and (0.9,0.3) has second version of H-GA a better SR than the first. It seems that the first version is also slightly better than the second version when comparing AES. Only in density-tightness combinations (0.3,0.7), (0.5,0.5) and (0.9,0.3) is the second version more efficient in AES than the first version. One must note, again, that in these combinations only a few solutions where found and therefore nothing really conclusive can be said about this difference.

8.3 About Arc-GA

For Arc-GA, the results in table 6 show that it performed best of all three GAs in density-tightness combinations (0.1,0.1) and (0.5,0.3). Again, the best performance in the first combination is shared with ESP-GA and the third version of H-GA. With regard to AES, Arc-GA performs best in density-tightness combinations (0.1,0.7), (0.3,0.5), (0.5,0.5), (0.7,0.3) and (0.9,0.3).

In general, `Arc-GA` does not perform as good as all other `GA`s. In general I can conclude that the SR of `Arc-GA` drops-off earlier and faster than the other methods. When looking at AES however, the opposite seems to be the case. It seems that when the `CSP`s get harder to solve, `Arc-GA` remains efficient with the number of evaluations it uses. Throughout the results table I can say that `Arc-GA` is the most efficient `GA`, its low SR however leads me to conclude that `Arc-GA` has the least performance of all three `GA`s.

9 Conclusion

The results in the result table (table 6) give some indication of what is called the *landscape of solvability* of the different `GA`s. This landscape of solvability is divided into three parts. The first part of the landscape is where, generally, all `CSP`s are solved by the `GA`s. All `CSP`s which are solved with a SR of 100% by all `GA`s fall into this area of the landscape of solvability. Generally, all these `CSP`s have many solutions.

The second part of the landscape of solvability is where `CSP`s can not be solved by the `GA`s. In part this is because these `CSP`s have no solution. In the result table this area is indicated by a SR of 0% for all `GA`s.

This leaves a third area of the landscape of solvability, generally called the *mushy region*. In this area, `CSP`s are hard to solve and often have only one or just a few solutions. It is this area, that is important when comparing different `GA`s. It is obvious that a `GA`, that can still solve these `CSP`s, is better than a `GA` that can not solve these `CSP`s. The 'mushy region' in the landscapes of solvability of the evaluated `GA`s are the `CSP`s with density-tightness combinations (0.1,0.9), (0.3,0.7), (0.5,0.5), (0.7,0.3) and (0.9,0.3) and to a lesser extend density-tightness combinations (0.1,0.7) and (0.3,0.5).

When looking at the mushy region of the landscape of solvability and comparing the SRs of the `GA`s I can conclude that `Arc-GA` performs the least of all `GA`s. Only in density-tightness combinations (0.1,0.7) and (0.3,0.7) does `Arc-GA` have a larger SR than `ESP-GA` although this difference is small. Compared to the third version of `H-GA` in density-tightness combination (0.3,0.7), `Arc-GA` still finds a few solutions while the third version does not.

`ESP-GA` has a mixed performance. In density-tightness combinations (0.1,0.7), (0.1,0.9), (0.3,0.5), (0.3,0.7) and (0.7,0.3), `ESP-GA` does not perform as well as the other `GA`s, always being out-performed by one or more `GA`s. In density-tightness combinations (0.5,0.5) and (0.9,0.3), the opposite is the case. In these two combinations `ESP-GA` performs best while these two combinations are hard to solve for all `GA`s. In general however I can conclude that, apart from these two exceptions, `ESP-GA` performs only slightly better than `Arc-GA`. I come to this conclusion based on the fact that `ESP-GA`, as does `Arc-GA` and to a lesser extend the third version of `H-GA`, fails to find some solutions relatively early and because for a large part of the mushy region, `ESP-GA` has a lesser performance than `H-GA` although it has a better SR than `Arc-GA`.

As already said earlier, the third version of `H-GA` fails to find any solution in density-tightness combination (0.3,0.7) while the other GAs still find some solutions. This failure to find solutions should not be over-stressed as the best performing GA in this density-tightness combination only finds a solution in a mere 5% of its runs. What I find more serious is that the third version of H-GA fails to find some solutions in density-tightness combinations (0.3,0.5) and (0.7,0.3). Furthermore the third version of H-GA performed less when compared to the first two versions of H-GA in density-tightness combinations (0.1,0.9) and (0.9,0.3). In general I conclude that the third version of H-GA is only slightly better or on par with ESP-GA but has a lesser performance than the first two versions.

The difference between the first and the second version of H-GA is small. In density-tightness combinations (0.1,0.9) and (0.5,0.5) the first version has a slightly better SR than the second version while in density-tightness combinations (0.3,0.9) and (0.9,0.3) the second version of H-GA is slightly better than the first version. When reviewing AES, the first version performs slightly better than the second version. Therefore I conclude that the first version H-GA performs slightly better that the second version.

From the difference between the versions of H-GA something else can be concluded. The main difference between the first two versions and the third version is that this last version uses both heuristic operators together while the first two versions only use one of the heuristics operators. This means that the heuristics of the third version of H-GA have a larger effect on the search process, as they are used more often. The results show that this increase of heuristics or the increase of strength of the heuristics used, did not result in a better performance of the GA with regard to SR, although it did increase the efficiency in which the GA found the solutions. This can be observed by the decrease in AES. This result supports, to a certain extend, the notion that by using stronger or more heuristics that performance of the GA is not increased. A possible explanation for this could be that the use of stronger heuristics make the GA more prone to premature convergence into local optima.

In general, I can conclude that the first two versions of H-GA perform better than the other GAs. I come to this conclusion mostly because these two GAs still solve CSPs with density-tightness combinations (0.1,0.7), (0.3,0.5) and (0.7,0.3) with a SR of 100% while maintaining a high SR throughout the rest of the mushy region, although other GAs, especially ESP-GA, have a better SR in some density-tightness combinations in the mushy region. It must also be noticed that the differences between the different GAs are in general small and it is because of this that this comparison can not be more clear and a definite best GA can not, in all fairness, be chosen.

10 Future Study

In this thesis I have compared three different GAs with each other. As is generally accepted in science in general and computer science especially, apart to answering some questions about a subject, many more arise that are interesting to study. Because of the limits that a comparison of the three GAs imposes on this thesis, I too have found some questions that have to remain unanswered for now but are interesting enough to warrant further study.

All three GAs still need more study, mostly where the use of different heuristics are involved. I have chosen to use these heuristics mostly for ease of comparison and to make sure that a comparison of these GAs is as fair as is possible. That does not mean that the heuristics that I have used are the best that can be found for use with the respective GAs. Further study about the effects of using different heuristics can result in a better understanding of the use of these heuristics. As can be seen in the result table, results of different GAs can lie very close to each other and further optimization of the use of heuristics may prove that the performance of a specific GA show so much improvement that in some instances the use of a specific heuristic is preferable over another. To further increase of knowledge about heuristics and their specific use, I suggest a quantitative study of their effects.

Although ESP-GA has performed about average, compared to the other GAs, I believe that it has a number of strong points that make further study of this specific GA, and the method used in it, interesting as I think that dramatic improvement in its performance is possible. The best example of such a study is the combination of ESP-GA, or more precisely the repair method that it uses, with other GAs. The method used in ESP-GA is fairly independent of the GA that it is incorporated to. This makes that ESP-GA can be combined with any of the GAs that I have studied in this thesis or with other GAs. I believe that this, 'Double Whammy'-strategy, can be very effective and therefore worth investigating. Another strong point of ESP-GA is its efficiency in solving non-hard CSPs. An interesting question that remains to be studied, is if the efficiency of ESP-GA can be combined with the solving power of other GAs or methods. One of ESP-GA's main weaknesses is the large amount of 'hidden work' it requires to solve a CSP. An interesting suggestion from E. Marchiori, to only repair a random subset of all constraints, may have interesting effects and may help to improve the time needed to solve a CSP.

An interesting notion can be detected when comparing Arc-GA and the three versions of H-GA. It seems that the more one uses heuristics in their method, the fewer evaluations to solution are necessary. The third version of H-GA uses both heuristic operators and has a better efficiency than the other two versions. Arc-GA adds to the two Arc-operators, that it uses a heuristic supported fitness function and performs in many cases better than version three of H-GA. I think that it is interesting to study at what point an increase of stronger or more heuristic operators in fact decrease performance of an GA, because of the limitation those operators lay on the diversity of the search, i.e. where does an increase of the strength of heuristics in the methods stops being a positive influence on the performance of that GA and when does it incur premature convergence.

I have found that many comparisons of different GAs using different methods for solving CSPs have been made. Most introduced a new method and compared it to one or two other methods. I believe that a study of all these methods is now needed, if only to answer that single question: what is the reason that one method outperforms another? I believe that instead of trying to expand the already broad array of different heuristics and genetic operators that exist in the field of solving CSPs (or COPs, for that matter) with one more method, more study should be devoted into comparing the existing methods to each other quantitative. With the information gained from these comparisons a more 'directed' search for better GA can then, again, be staged.

A Sample output of the random CSP instance generator

```
5                 | The number of
                  |  variables: n = 5
                  |
55555             | The domain sizes
                  |  of the variables:
                  | m = 5
                  |
0  0  0  0  0  \  |
0  0  0  0  0  |  |
0  0  0  0  0  >  | C1,2
0  0  0  0  0  |  |
0  0  0  0  0  /  |
0  0  0  0  0  \  |
0  0  0  0  0  |  |
0  0  0  0  0  >  | C1,3
0  0  0  0  0  |  |
0  0  0  0  0  /  |
0  0  0  0  0  \  |
0  0  0  0  0  |  |
0  0  0  0  0  >  | C1,4
0  0  0  0  0  |  |
0  0  0  0  0  /  |
0  0  0  0  0  \  |
0  0  0  0  0  |  |
0  0  0  0  0  >  | C1,5
0  0  0  0  0  |  |
0  0  0  0  0  /  |
1  0  1  0  1  \  |
0  1  0  0  1  |  |
0  1  1  0  0  >  | C2,3
1  0  0  0  0  |  |
0  0  1  1  0  /  |

0  0  0  0  0  \  |
0  0  0  0  0  |  |
0  0  0  0  0  >  | C2,4
0  0  0  0  0  |  |
0  0  0  0  0  /  |
0  0  0  0  1  \  |
0  1  1  0  0  |  |
0  1  0  1  1  >  | C2,5
0  1  0  0  1  |  |
0  0  0  0  1  /  |
0  1  0  0  0  \  |
0  1  0  0  1  |  |
0  1  0  1  0  >  | C3,4
1  1  1  0  0  |  |
0  1  0  0  0  /  |
0  0  1  0  1  \  |
0  1  0  0  0  |  |
0  1  1  1  1  >  | C3,5
1  0  0  1  0  |  |
1  1  0  1  0  /  |
1  0  1  1  1  \  |
0  1  1  0  0  |  |
1  0  0  0  0  >  | C4,5
1  1  1  1  1  |  |
1  1  0  1  1  /  |
```

B Graphical representation of the result table, SR and AES

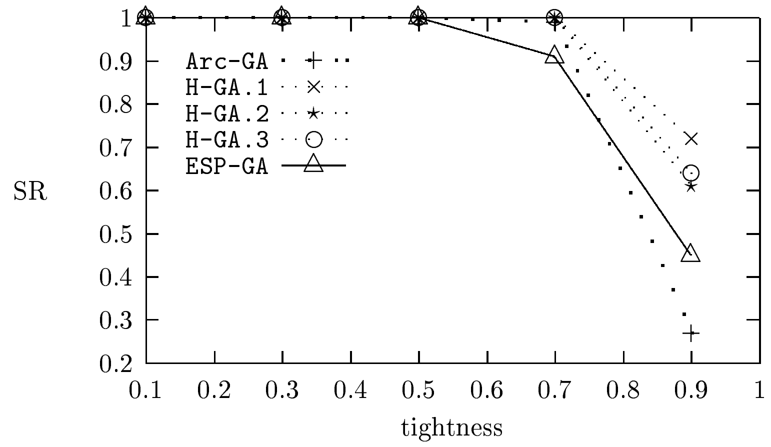

Figure 4: SR-Graph of all three methods with density 0.1

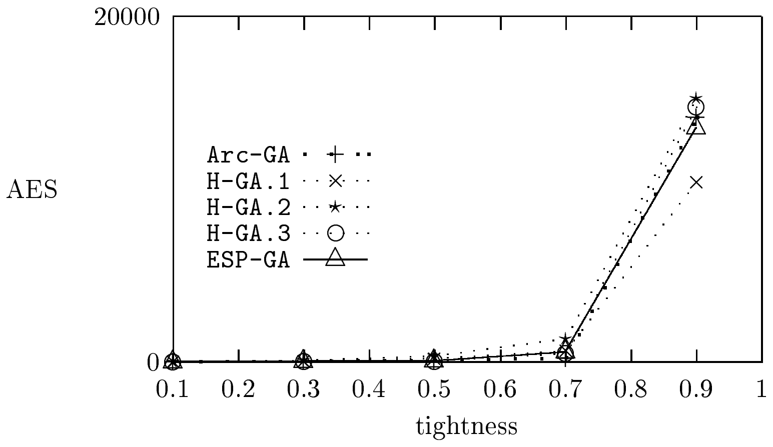

Figure 5: AES-Graph of all three methods with density 0.1

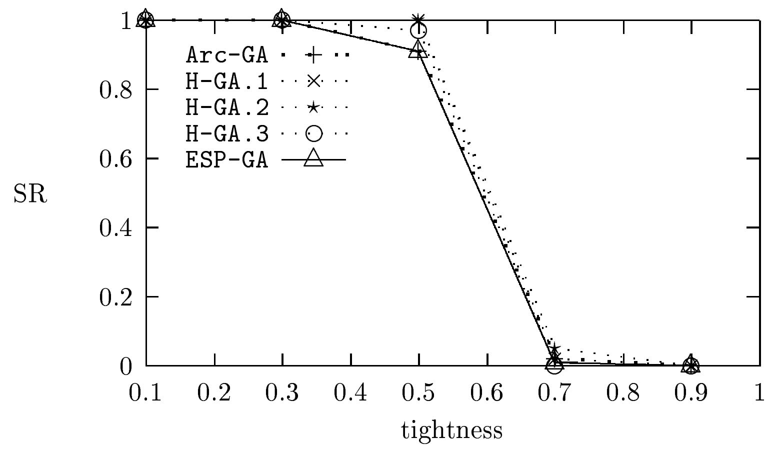

Figure 6: SR-Graph of all three methods with density 0.3

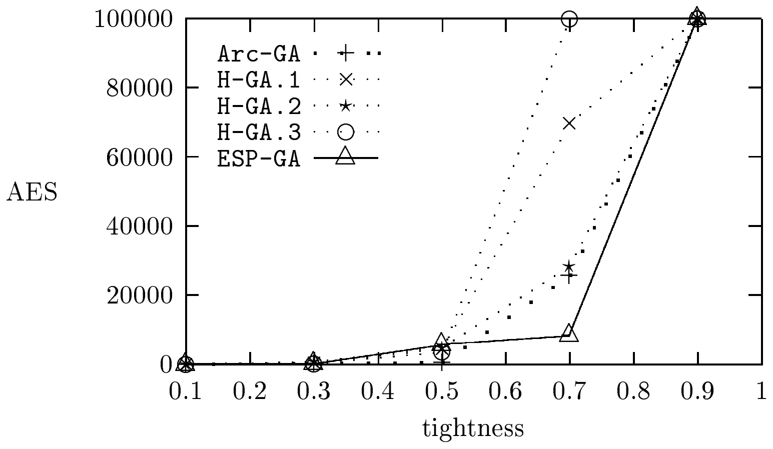

Figure 7: AES-Graph of all three methods with density 0.3

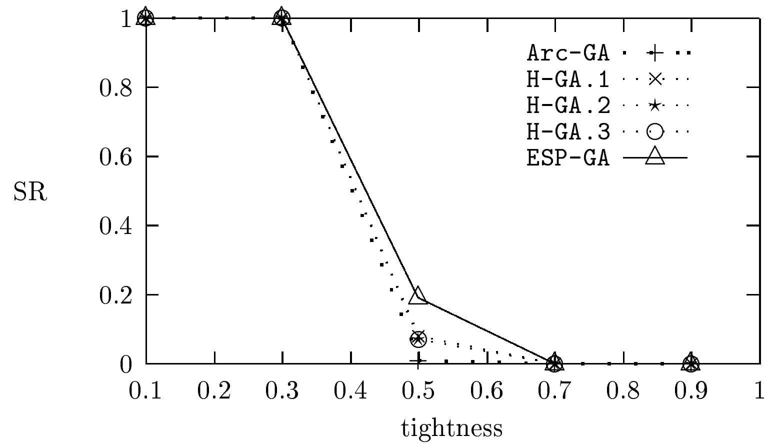

Figure 8: SR-Graph of all three methods with density 0.5

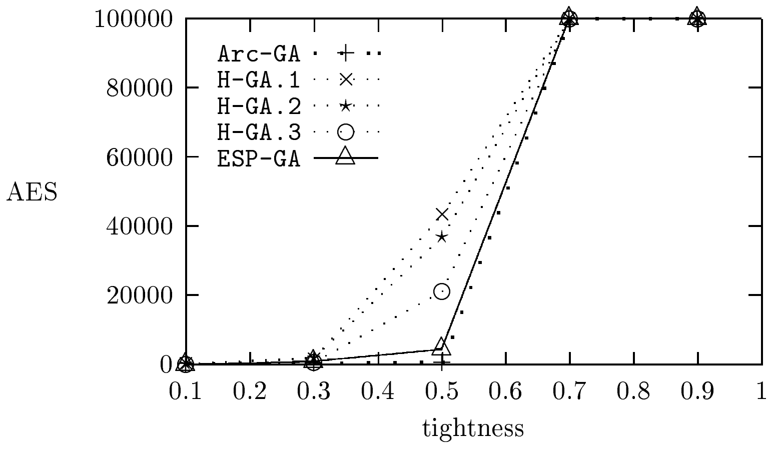

Figure 9: AES-Graph of all three methods with density 0.5

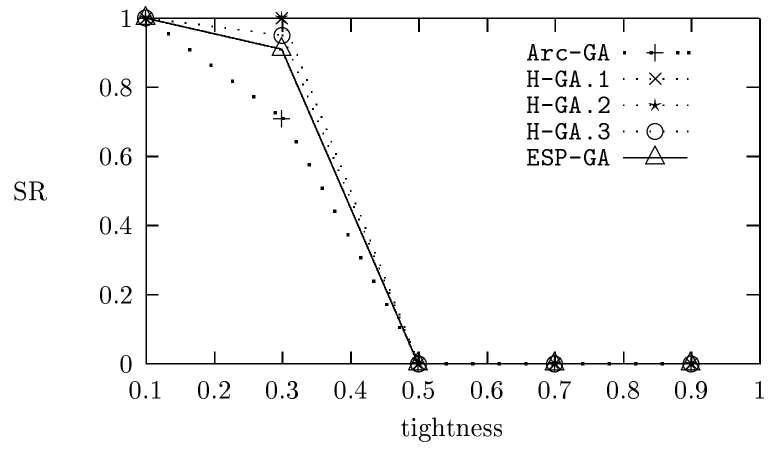

Figure 10: SR-Graph of all three methods with density 0.7

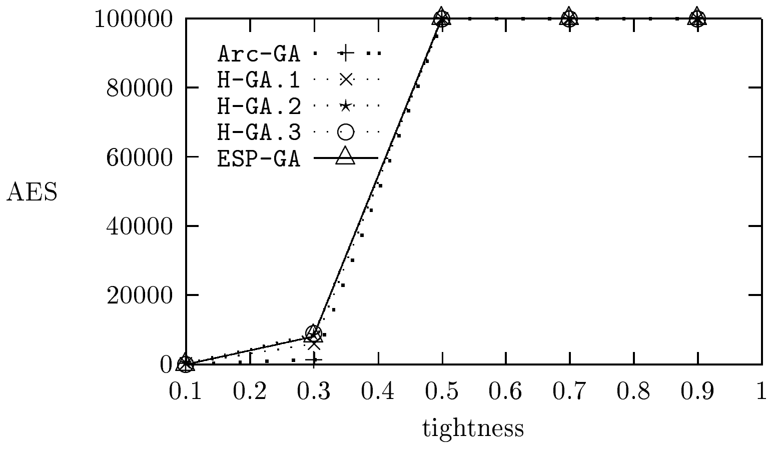

Figure 11: AES-Graph of all three methods with density 0.7

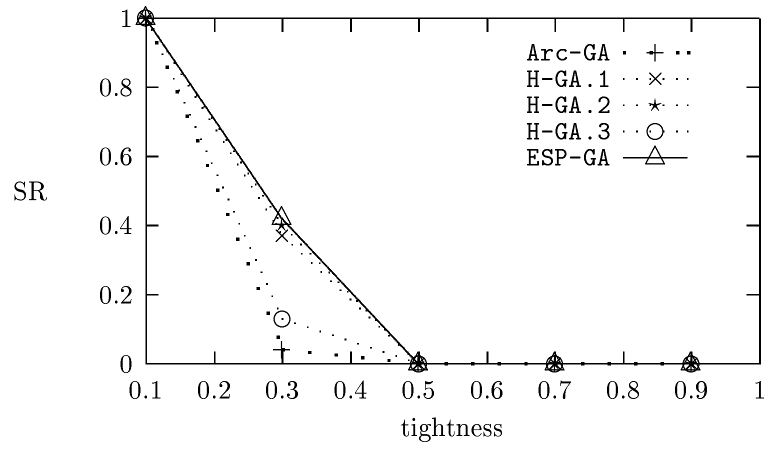

Figure 12: SR-Graph of all three methods with density 0.9

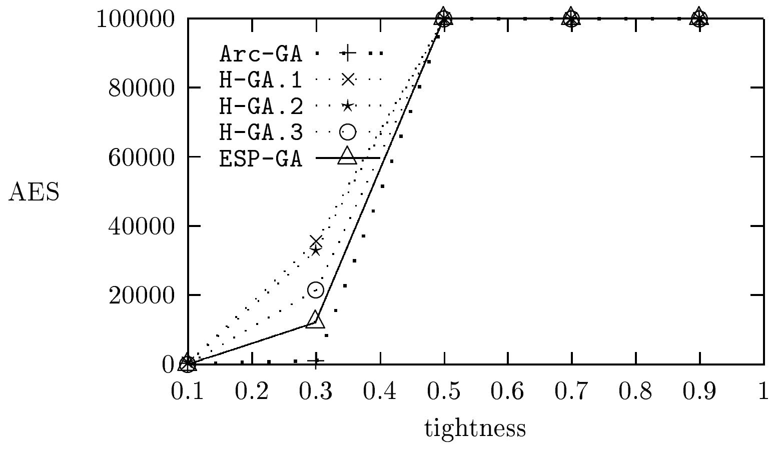

Figure 13: AES-Graph of all three methods with density 0.9

C Graphical representation of landscapes of solvability of all three methods, SR and AES

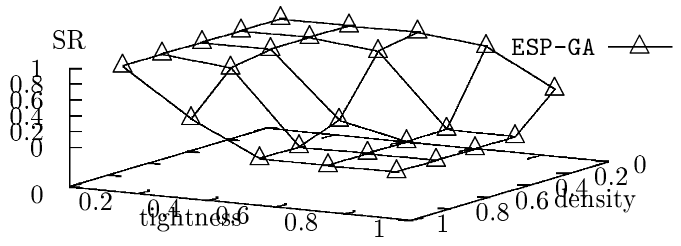

Figure 14: SR-Graph of landscape of solvability of ESP-GA

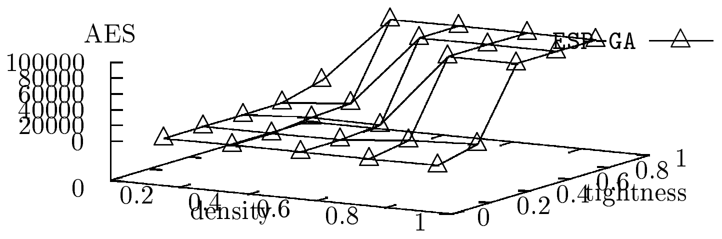

Figure 15: AES-Graph of landscape of solvability of ESP-GA

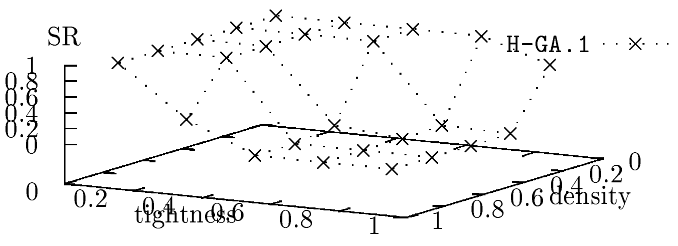

Figure 16: SR-Graph of landscape of solvability of H-GA.1

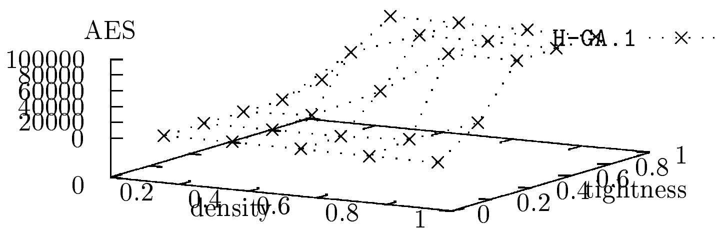

Figure 17: AES-graph of landscape of solvability of H-GA.1

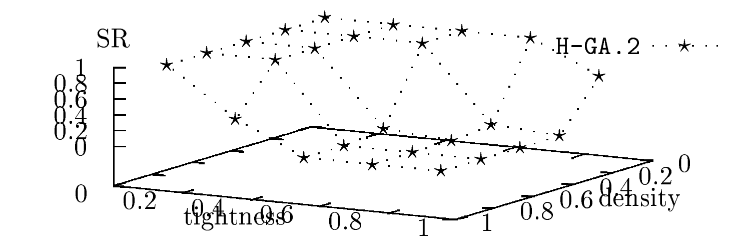

Figure 18: SR-Graph of landscape of solvability of `H-GA.2`

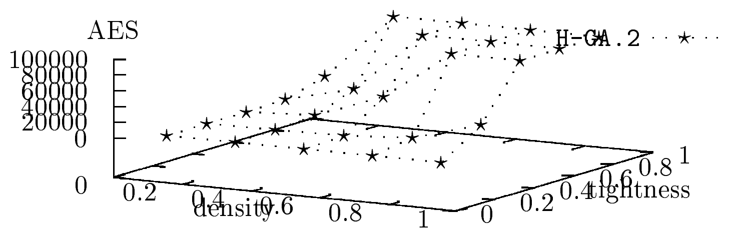

Figure 19: AES-Graph of landscape of solvability of `H-GA.2`

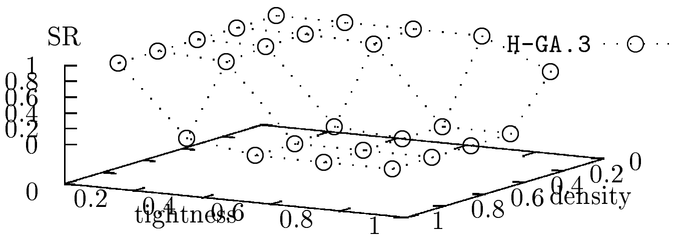

Figure 20: SR-Graph of landscape of solvability of `H-GA.3`

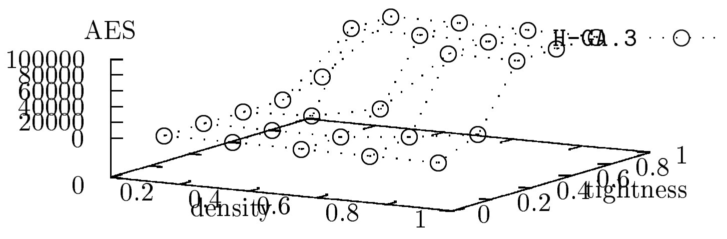

Figure 21: AES-Graph of landscape of solvability of `H-GA.3`

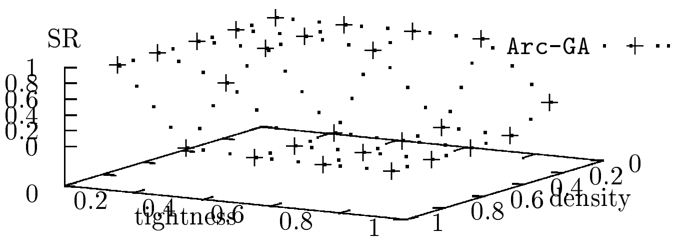

Figure 22: SR-Graph of landscape of solvability of `Arc-GA`

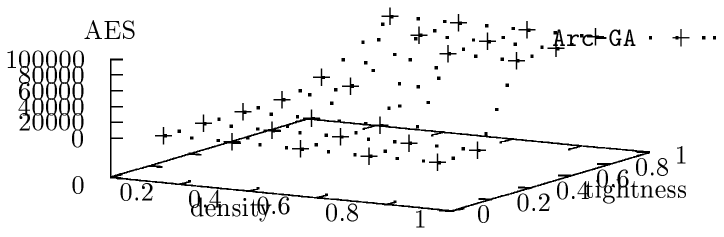

Figure 23: AES-Graph of landscape of solvability of `Arc-GA`

References

[1] J. Bowen and G. Dozier. Solving constraint satisfaction problems using a genetic/systematic search hybride that realizes when to quit. In Eshelman [15], pages 122–129.

[2] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th IJCAI-91*, volume 1, pages 331–337. Morgan Kaufmann, 1991.

[3] R. Dechter. Constraint networks. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley-Interscience, 1992.

[4] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithms. In IEEE [17], pages 306–311.

[5] 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 Press, 1995.

[6] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Heuristic genetic algorithms for constrained problems; part 1: Principles. Technical Report IR-337, Vrije Universiteit Amsterdam, December 1993.

[7] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In IEEE [17], pages 542–547.

[8] 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.

[9] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. GA-easy and GA-hard constraint satisfaction problems. In M. Meyer, editor, *Proceedings of the ECAI-94 Workshop on Constraint Processing*, number 923 in Lecture Notes in Computer Science, pages 267–284. Springer-Verlag, 1995.

[10] 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.

[11] A.E. Eiben and Zs. Ruttkay. Constraint satisfaction problems. In Th. Bäck, D. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Algorithms*, pages C5.7:1–C5.7:8. IOP Publishing Ltd. and Oxford University Press, 1997.

[12] 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 LNCS, pages 95–106. Springer, Berlin, 1997.

[13] A.E. Eiben and J.K. van der Hauw. Solving 3-SAT with adaptive Genetic Algorithms. In IEEE [18], pages 81–86.

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

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

[16] E.C. Freuder. The many paths to satisfaction. In M. Meyer, editor, *Constraint Processing*, volume LNCS 923, pages 103–119. Springer-Verlag, 1995.

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

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

[19] A. K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence.* Wiley-Interscience, 1992.

[20] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In Th. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 330–337. Morgan Kaufmann, 1997.

[21] Z. Michalewicz. Genetic algorithms, numerical optimization, and constraints. In Eshelman [15], pages 151–158.

[22] 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. MIT Press, 1995.

[23] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs.* Springer, Berlin, 3rd edition, 1996.

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

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

[26] J. Paredis. Genetic state-space search for constrained optimization problems. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI 93)*, pages 967–972. Morgan Kaufmann, 1993.

[27] J. Paredis. Co-evolutionary 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 46–56. Springer-Verlag, 1994.

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

[29] Jan Paredis. Coevolving cellular automata: Be aware of the red queen. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

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

[31] M.C. Riff-Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves CSP. In IEEE [18], pages 279–284.

[32] M.C. Riff-Rojas. Evolutionary search guided by the constraint network to solve CSP. In IEEE [18], pages 337–348.

[33] 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.

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

[35] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

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

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