VRIJE UNIVERSITEIT

Solving Constraint Satisfaction Problems with Evolutionary Algorithms

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 24 november 2005 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Bartolomeus Gerardus Wilhelmus Craenen

geboren te Den Helder

promotor: prof.dr. A.E. Eiben

To family, friends, and books . . .

Samenvatting

Het oplossen van Constraint Satisfaction problemen door evolutionaire algoritmen.

Constraint Satisfaction problemen worden gedefiniëerd door variabelen, domeinwaarden die aan deze variabelen toegekend kunnen worden en beperkingen (*constraints*) die bepalen welke domeinwaarden aan welke variabelen toegekend mogen worden. Een oplossing voor een *Constraint Satisfaction* probleem bestaat uit de toekenning van domeinwaarden aan alle variabelen op zodanige wijze dat geen van de beperkingen geschonden wordt.

Evolutionaire algoritmen zijn modellen die, met behulp van de rekenkracht van een computer, problemen oplossen aan de hand van de Darwinistische evolutieleer. Evolutionaire algoritmen behoren tot de klasse van non-deterministische algoritmen en ontwikkelen een oplossing van een probleem uit een willekeurig bepaalde initiële populatie van partiële oplossingen, gebruikmakende van natuurlijke selectie en kansbepaalde reproductie en mutatie. Het is onze stelling dat evolutionaire algoritmen voor alle probleemtypen een alternatieve oplossingsmethode zijn. Voor een aantal probleemtypen is dit al aangetoond. In dit proefschrift wordt getoetst of dat ook voor *Constraint Satisfaction* problemen het geval is. We doen dit door een evolutionair algoritme te ontwerpen dat in effectiviteit en efficiëntie superieur is aan alle tot dusver gepubliceerde evolutionaire algoritmen. De effectiviteit is gedefiniëerd als het oplossend vermogen van het algoritme terwijl de efficiëntie de benodigde hoeveelheid werk tot het vinden van een oplossing tot uitdrukking brengt. Door de effectiviteit en de efficiëntie te vergelijken met alternatieve oplossingsmethoden kan onze stelling gestaafd worden.

Uit onze bevindingen blijkt dat qua effectiviteit ons evolutionaire algoritme vergelijkbaar is met alternatieve oplossingsmethoden, maar het qua efficiëntie minder goede resultaten laat zien. Gezien deze resultaten luidt de eindconclusie van het proefschrift dan ook dat wanneer alleen de effectiviteit van evolutionaire algoritmen van belang is, evolutionaire algoritmen een vergelijkbare alternatieve oplossingsmethode kunnen zijn. Als echter ook de efficiëntie van evolutionaire algoritmen in ogenschouw genomen wordt, is dit in mindere mate het geval. Behalve pure prestatie kunnen echter ook andere eigenschappen bij de beoordeling een rol spelen. Zo zijn evolutionaire algoritmen eenvoudig te ontwerpen en kunnen ze met weinig aanpassingen ook op andere probleemtypen toegepast worden. Daar staat tegenover dat evolutionaire algoritmen, als onderdeel van de klasse van non-deterministische algoritmen, niet compleet zijn en als zodanig het vinden van een oplossing niet kunnen garanderen.

In dit proefschrift wordt een aantal bijdragen gepresenteerd die de directe toepassing binnen dit onderzoek ontstijgen en de wetenschap in het algemeen en het experimenteel onderzoek naar evolutionaire algoritmen in het bijzonder, ten goede komen. Dit zijn:

- een methodologie voor het construeren van een test-set van *Constraint Satisfaction* problemen, specifiek voor het experimenteel onderling vergelijken van de prestaties van non-deterministische algoritmen in het algemeen en evolutionaire algoritmen in het bijzonder;
- een overzicht van acht eerder gepubliceerde algoritmen voor het oplossen van *Constraint Satisfaction* problemen, inclusief volledige beschrijving van de gebruikte technieken alsmede experimentele resulaten voor het bepalen van hun relatieve prestaties;
- een methodologie voor het vergelijken en rangordenen van de prestaties van evolutionaire algoritmen, gebruikmakende van eerder gedefiniëerde meetmethoden, relatieve vergelijkingen in het effectiviteit-efficiëntie vlak en statistische analyse;
- de notie van het *memetic overkill*-effect, en een methodologie voor het vaststellen van *memetic overkill* in evolutionary algoritmen door de-evolutie van het algoritme;
- een software platform voor experimenteel onderzoek naar evolutionaire algoritmen waarin de algoritmen uit het overzicht op een uniforme manier zijn geïmplementeerd.
- de vaststelling van het best presterende evolutionare algoritme voor het oplossen van *Constraint Satisfaction* problemen.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Every day life is filled with limitations; constraints. A day still has only 24 hours and it is impossible to be in more than one place at the same time. Coping with constraints is therefore something that is inherent to coping with life itself. As a result, it should come as no surprise that solving constrained problems in one shape or another is also an inherent part of science. Whatever the origin of the constraints, be it physical, social or or otherwise, a constrained problem is only solved if all constraints are satisfied.

Constrained problems can be divided into two classes: Constrained Optimising Problems (COPs) and constraint satisfaction problems (CSPs) [27]. The difference between these classes is that in the first an optimal solution that satisfies all constraints should be found, while in the second class any solution will do.

These two classes are closely related. The difference between the two is that, in addition to constraints, constrained optimisation problems also define an optimisation function, often expressing the cost of getting to a solution. When all solutions of the constraint satisfaction problem can be found, they can be ordered using this function. By selecting the optimal solution, the constrained optimising problem is also solved. It is for this reason that the constraint satisfaction problem is often seen as a sub-class of the constrained optimising problem.

In Table 1.1, the relationship between problems having an objective function, constraints or both is shown ([32]). FOP stands for Function Optimisation Problem. Problems without an objective function and constraints remain undefined in this context.

		Constraints	
		Yes	*No*
Objective	*Yes*	COP	FOP
Function	*No*	CSP	undefined

Table 1.1: Problems having an objective function, constraints. or both.

In Evolutionary Computation, constrained problems were studied right from the beginning. This came about by the realisation that evolution has shown itself to be a robust optimiser in constrained environments. If evolution in the complex environment of nature can find an optimal solution, surely an evolutionary algorithm should be able to do the same in a computational environment of lesser complexity. Unfortunately, the early results were disappointing. The operators used at that time were blind to constraints and overall efficiency was low. This sparked an interest in designing specific genetic operators, representations and fitness functions that can handle constrained problems.

1.1 Constraint Satisfaction Problems

A commonly used example of a constraint satisfaction problem is the N-queens problem. The N-queens problem features a chess-board of $N \times N$ squares using N queens as pieces. As in chess, queens threaten other pieces horizontally, vertically and diagonally. The objective of the game is to place *all* queens on the board so that they do not threaten each other. Figure 1.1 shows a solution of the 8-queens problem.

The N-queens problem is a constraint satisfaction problem because it restricts the placement of the queens to non-threatened squares and all solutions of the problem are equally valid. The constraints defined by the N-queens problem are:

1. No two queens may be placed in the same row;
2. No two queens may be placed in the same column;
3. No two queens may be placed diagonally from each other.

Some definitions of the N-queens problem include a fourth constraint that two queens may not occupy the same square on the game-board even though this is implied by the constraints given above.

Many constraint satisfaction problems have been identified, in fact the number of different constraint satisfaction problems that can be studied is infinite. A general mathematical description will be formulated to describe all constraint satisfaction problems. A study of all possible constraint satisfaction problems is outside the scope of this thesis however. We restrict the current investigation as follows:

1. Only *binary* constraint satisfaction problems are studied in this thesis. A binary constraint satisfaction problem defines constraints as a relationship between only two entities. The N-queens problem is an example of a binary constraint satisfaction problem. All constraints define a relationship between two queens. Theoretically, all non-binary constraint satisfaction problems can be transformed into a binary constraint satisfaction problem [83].
2. Only constraint satisfaction problems with equal domains for each variable are studied in this thesis. Again, the N-queens problem is a good example of such a

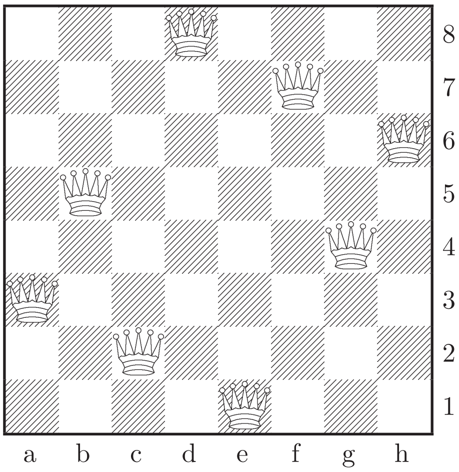

Figure 1.1: A solution of the 8-queens problem.

problem. The game-board of the N-queens problem is a square. All queens have the same number of locations they can be placed at. The locations themselves are also discrete: there are only a finite number of possibilities. A constraint satisfaction problem with both restrictions is called a constraint satisfaction problem with *discrete uniform domain sizes*. Any constraint satisfaction problem with non-uniform domain sizes can be transformed to a uniform domain size constraint satisfaction problem and a continuous constraint satisfaction problem can be approximated by a discrete constraint satisfaction problem, theoretically with infinite accuracy.

3. Only *randomly generated* constraint satisfaction problems will be studied in this thesis. We only use randomly generated constraint satisfaction problems because of two reasons:

   (a) A thorough investigation on the constraint satisfaction problem necessitates the use of a large number of problem instances with varying but specific complexity parameters. The best way to obtain these problem instances is to use a constraint satisfaction problem generator.

   (b) An accurate investigation on the constraint satisfaction problem necessitates the use of problem instances with the least amount of bias or unknown properties or irregularities. The best way to obtain these problem instances is to generate them randomly.

   Alternatives to using constraint satisfaction problem instances generated randomly by a problem generator is using problem instances constructed by hand or problem instances derived from constraint satisfaction problems occurring in the real world. Both alternatives however are either not capable of providing enough problem instances or are not able to provide problem instances without bias, irregularities or unknown properties.

1.2 Evolutionary Algorithms

Evolutionary algorithms are the subject of a research field called Evolutionary Computation. Although the term was invented as recently as 1990, the field has a history that spans over four decades [38]. In the 1950s and '60s, many independent efforts were devoted to simulate evolution on a computer but only four avenues of investigation have survived as main disciplines in the field: evolutionary strategies, evolutionary programming, genetic algorithms, and genetic programming. The differences between these four disciplines are characterised by the typical application areas, data representations, the methods for producing random variance in the population, and the method employed for selecting parents and offspring.

Evolutionary algorithms incorporate the metaphor of Darwinian evolution. In "The Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life" [21], C. Darwin described evolution as a two-step process of random variation and selection. A population of individuals is exposed to an environment and responds with a collection of behaviours. Some of these behaviours are better suited to meet the demands of the environment than others, selection then tends to eliminate those individuals that demonstrate inappropriate behaviours. The survivors then reproduce and their traits are passed on to their offspring. Replication of the individuals is never without error, nor can the individual's traits remain free of random mutations. Introduction of random variation in turn leads to novel traits. Over successive generations, increasingly more appropriate behaviours accumulate within evolving ancestral families [62, 5].

Evolutionary algorithms capture evolution by modelling it algorithmically and simulating it on a computer. The most elementary of models takes a population of individuals and randomly varies all individuals according to rules expressed in what are called variation operators. Then, based on an objective function, each individual in the population is assigned a value expressing how close it is to some solution of the problem that is investigated. This value is called the fitness of the individual. Based on these fitness values a selection of individuals is used in the next iteration of the problem.

Evolutionary algorithms offer a powerful alternative to a wide variety of traditional problem-solving techniques. Because the relationship between the algorithm and the problem is captured in the objective (fitness) function, they usually do not require any in-depth mathematical understanding of the problem itself. Evolutionary algorithms are also capable of efficiently handling problems with many variables or that have frequently and unpredictably moving objectives. Evolutionary algorithms, because of their stochasticity, are very robust and can cope well with noisy, inaccurate and incomplete data. Furthermore, they are relatively easy to hybridise with other techniques and adapt well to changing priorities in the problem by simply changing the weights in the objective function. Because evolutionary algorithms are modular, the evolutionary mechanism is separate from the problem representation, they can be transferred from problem to problem and are therefore relatively cheap and quick to implement. The open design of an evolutionary algorithm allows for the incorporation of arbitrary constraints, simultaneous multiple objectives and the mixing of continuous and discrete parameters.

1.3 Motivation and Main Goal

The main motivation for writing this thesis is that we believe that for many problems, evolutionary computation can provide a viable alternative to other algorithms. Other studies have already shown that this is true for a number of problems. In this thesis we investigate if this is the case for the constraint satisfaction problems.

We intend to test the viability of using evolutionary computation to solve the constraint satisfaction problem by constructing the best possible evolutionary algorithm for solving this problem and comparing its performance to alternative techniques. This then is the main goal of the thesis.

We choose the constraint satisfaction problem because solving these kinds of problems is especially challenging for evolutionary algorithms. The constraint satisfaction problem is hard to solve for evolutionary algorithms because of the absence of an objective function to optimise. Moreover, some very effective and efficient classical algorithms have been found for solving them, so there is strong competition.

In the last two decades much effort was put in solving constraint satisfaction problems with evolutionary algorithms. This resulted in a large number of evolutionary algorithms, some of which are closely related to each other. We intend to base the design of the superior evolutionary algorithm on these earlier introduced algorithms, by including an inventory of these algorithms and the techniques they use and comparing and analysing their performance.

Unfortunately, the evolutionary algorithms were run on different constraint satisfaction problem test-sets, making comparison between them difficult. Moreover, some of these test-sets were found to be deficient in some way. Constraint satisfaction problem research also made important progress during this period, especially in generating random constraint satisfaction problem test-sets and in complexity measures. A thorough investigation into the viability of evolutionary algorithms for solving constraint satisfaction problems has to take this into account as well.

1.4 Technical Objectives of the Thesis

From the main goal the following technical objectives for the thesis can be derived:

1. Construct and analyse a test-set of constraint satisfaction problem instances for evolutionary algorithms to solve. The test-set, the generator models and the

classical algorithms used to generate the test-set will be made available for other researchers.

2. Provide a comprehensive inventory of evolutionary algorithms for solving constraint satisfaction problems. To reduce the influence of different programming languages and programming styles, all algorithms in the inventory will be re-implemented in a single library. This library will also be made available.
3. Compare the performance of the evolutionary algorithms in the inventory to each other. The comparison will be based on a number of both traditional and new measures.
4. Identify which algorithms have the best performance and identify which techniques in these algorithms cause better performance. Determine the balance between the techniques used and the evolutionary components of these algorithms.
5. Increase the performance of an existing evolutionary algorithm by designing a variant which uses the lessons learned and compare the performance of this algorithm with the performance of classical algorithms. The variant is included in the library as well.

The most important contribution to the scientific community made by this thesis will be the superior evolutionary algorithm for solving the constraint satisfaction problem. The superior performance of this algorithm is based on a solid justification using a comprehensive experimental methodology that is also of value to the community. This methodology spans the whole experimental track; using a newly constructed test-set of constraint satisfaction problem instances, traditional and new performance measures that are explicitly defined, an inventory identifying effective algorithms over less effective ones, and different methods for comparing the performance of evolutionary algorithms. Some parts of the methodology are specific for the constraint satisfaction problem but with some alteration can be generalised for use with related problems like the satisfiability problem or graph colouring. Other parts, however, are useful for the scientific community in general; especially the new performance measures and the methodology for analysing the performance of the algorithms.

1.5 Overview of the Thesis

The thesis in structured in the following way.

In the next chapter, the constraint satisfaction problem is defined. These definitions will be used throughout the rest of the thesis. Using this definition, a number of complexity measures are defined. The chapter is concluded with a description of six random constraint satisfaction problem instance generators.

In Chapter 3 two classical algorithms for solving the constraint satisfaction problem will be described. These algorithms will be used to calculate the complexity of generated constraint satisfaction problem instances. They will also be used for a comparison of the performance of the evolutionary algorithms later on in the thesis.

In Chapter 4 the constraint satisfaction problem test-set is generated. The method used for generating the test-set is described in detail. The test-set is used throughout the rest of the thesis.

Chapter 5 introduces evolutionary algorithms as a part of the iterated local-search class of algorithms. Two other iterated local-search algorithms are also introduced: the Random Search algorithm and the Hill Climber algorithm. A canonical evolutionary algorithm for solving the constraint satisfaction problem is introduced as well: the *Intuitive Evolutionary Algorithm*.

Chapter 6 introduces the performance measures used to compare the algorithms in the thesis. The measures are then used to compare the performance of the three algorithms introduced in Chapter 5. The comparison is based on experiments using the test-set generated in Chapter 4.

An inventory of eight evolutionary algorithms for solving the constraint satisfaction problem is presented in Chapter 7. Each section of the inventory describes a single algorithm and includes parameter and characteristics tables for easy reference. The results of experiments are shown and discussed as well. The experiments use the test-set generated in Chapter 4.

Chapter 8 contains a comparison of the results of the experiments from Chapter 7. The results are compared separately for each performance measure, relative in the effectivity-efficiency plane, and ranked by statistical analysis. The comparison and ranking are used as a basis for drawing some preliminary conclusions.

Chapter 9 discusses the relative importance of the evolutionary components of natural selection and population of the four best performing algorithms selected through comparison in Chapter 8. Three of the four algorithms are found to suffer from memetic overkill. The remaining algorithm is adjusted to create the superior evolutionary algorithm. It too is checked to see if it suffers from memetic overkill.

The conclusion chapter of the thesis summarises the work done in the thesis and identifies the main contributions it makes to the scientific community. The performance of the superior evolutionary algorithm is compared to the performance of the alternative techniques introduced in Chapters 3 and 5. This rounds off the main goal of the thesis and checks whether our belief in evolutionary algorithms as described in the motivation for writing the thesis is correct.

Chapter 2

The Theory of Constraint Satisfaction Problems

In this chapter a formal definition of the constraint satisfaction problem is given. This definition is used throughout the rest of the thesis. Also introduced are complexity measures of the constraint satisfaction problem as well as ways of representing the constraint satisfaction problem in both matrices and graphs. Finally, different methods for generating binary constraint satisfaction problem instances randomly are described. Throughout the chapter, the N-queens problem is used as an example.

2.1 A Definition of the Constraint Satisfaction Problem

The introduction chapter of this thesis introduced the constraint satisfaction problem informally as a set of variables and a set of constraints between these variables. Variables are only assigned values from their respective domains and a solution of the constraint satisfaction problem was defined as the assignment of a value to all variables in such a way that no constraint would be violated. This section restates this definition more formally, based for a large part on the definition given in E. Tsang's standard work: "Foundations of Constraint Satisfaction"[83].

Each variable in a constraint satisfaction problem has a domain of possible values, and can only be assigned a value from that domain.

Definition 2.1 (Domain of a variable)
The ***domain of a variable*** *is a set of all possible values that can be assigned to that variable. If x is a variable, then D_x is used to denote its domain.*

Assigning a value to a variable is called labelling a variable. The number of variables and the size of the domains of these variables are parameters of the constraint satisfaction problem.

Definition 2.2 (Label)
Given a variable x with domain D_x. A label $\langle x, v\rangle$ *is then a variable-value pair representing the assignment of $v \in D_x$ to x.*

Labelling a number of variables with values simultaneously is done by a compound label.

Definition 2.3 (Compound label)
*Given variables x_i with domains D_{x_i}, with $i = 1, \ldots, n$, a **compound label** $L = (\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle)$ is then the simultaneous assignment of values $v_i \in D_{x_i}$ to a (possibly empty) finite set of variables. A compound label restricts labelling of a variable to only a single value: $\langle x_i, v_i\rangle \in L \wedge \langle x_i, v_j\rangle \in L \Rightarrow v_j = v_i$.*

The parenthesis notation for compound labels is used to distinguish them from a set of labels, note also that the labels in a compound label are not separated by commas.

To denote how many variables are labelled by a compound label we introduce the k-compound label.

Definition 2.4 (k-compound label)
*A k-**compound label** is a compound label which assigns values to k variables simultaneously. k is called the **arity** of the compound label.*

Definition 2.5 (Variable set of a compound label)
*The **variable set of a compound label** is the set of all variables that appear in the compound label.*

$$S_{(\langle x_1,v_1\rangle\langle x_2,v_2\rangle\ldots\langle x_k,v_k\rangle)} = \{x_1, x_2, \ldots, x_k\}$$

A compound label with smaller arity can be projected on a compound label with larger arity if all labels in the smaller compound label are part of the larger compound label.

Definition 2.6 (Projection of a compound label)
*Given compound label L and variable set S, the **projection of** L **to** S is $L \restriction S$ where $\langle x, v\rangle \in L \restriction S$ if and only if $x \in S$ and $\langle x, v\rangle \in L$.*

Constraints define relationships between sets of variables in a CSP.

Definition 2.7 (Constraint, variable set of a constraint)
Given compound labels L and L', a constraint c is a set of compound labels where $\forall L, L' \in c : S_L = S_{L'}$, $\forall L \in c : S_L \subseteq S$, $\forall L' \in c : S_{L'} \subseteq S$ and $\forall L \in c : S_c = S_L$.

The size of the variable set over which a constraint is defined is called the *arity* of a constraint.

Definition 2.8 (Arity of a constraint)
*Given a constraint c, with variable set S, the **arity** of c is equal to the size of S: arity$(c) = |S_c|$.*

If a variable is in the variable set of a constraint, it is said to be relevant to the constraint.

Definition 2.9 (Relevant variable to a constraint)
Given a constraint c, defined over variable set S, then variable x ***is relevant to*** *c if $x \in S_c$.*

A constraint is either violated or satisfied by a compound label. *Violating* a constraint is the opposite of *satisfying* a constraint. Although it is unnecessary to define *violates* explicitly, the term is commonly used in literature and the definition is added for convenience.

Definition 2.10 (Satisfies)
Given constraint c, defined over variable set S and compound label L with variable set S_L. If $S_c = S_L$ then L ***satisfies*** *c if and only if L is an element of c:*

$$\textit{satisfies}(L, c) \Leftrightarrow L \in c$$

If $S_c \subsetneq S_L$ then L ***satisfies*** *c if and only if the projection of L to S_c is an element of c:*

$$\textit{satisfies}(L, c) \Leftrightarrow L \upharpoonright S_c \in C$$

Definition 2.11 (Violates)
A compound label L ***violates*** *constraint c when it does not satisfy it:*

$$\textit{violates}(L, c) \not\Leftrightarrow \textit{satisfies}(L, c)$$

A compound label that violates a constraint is called a *conflict*.

The maximum number of compound labels that a constraint c can hold is the product of the domain sizes of all variables $x \in S_c$, where S_c is the variable set of c.

If a constraint contains the maximum number of compound labels it is called *non-restrictive*, as all possible compound labels satisfy the constraint. A constraint that does not contain the maximum number of compound labels is consequently called a *restrictive* constraint.

Using the definitions above the constraint satisfaction problem can be defined.

Definition 2.12 (Constraint Satisfaction Problem (CSP))
A ***constraint satisfaction problem*** *is a triple: $\langle X, D, C \rangle$, where:*

$X =$ *a finite set of* ***variables*** $\{x_1, x_2, \ldots, x_{|X|}\}$*;*

$D =$ *a function which maps every variable in* X *to a finite set of objects of arbitrary type:*

$$D : X \rightarrow \textit{finite set of objects (of any type)}$$

Take D_x *as the set of object mapped from* x *by* D*. These objects are called possible* ***values*** *of* x *and the set* D_x *the* ***domain*** *of* x*;*

$C =$ *a finite (possible empty) set of restrictive* ***constraints*** *on an arbitrary subset of variables in* X*. In other words,* C *is a set of sets of compound labels.*

We will use CSP to abbreviate constraint satisfaction problem.

We assume that two constraints in a CSP can not share the same variable set: if $\langle X, D, C\rangle$ is a CSP then $\forall c_1, c_2 \in C : S_{c_1} \neq S_{c_2}$.

The arity of a constraint satisfaction problem is the maximum arity of its constraints.

Definition 2.13 (Arity of a CSP)
Given constraint satisfaction problem $\langle X, D, C\rangle$*, the* ***arity*** *of that constraint satisfaction problem is defined as:*

$$\textit{arity}(\langle X, D, C\rangle) = \textit{max}\{arity(c) | c \in C\}$$

A solution of a constraint satisfaction problem is a k-compound label, where $k = |X|$, that satisfies all constraints of the constraint satisfaction problem.

Definition 2.14 (Solution of a CSP)
Given a constraint satisfaction problem $\langle X, D, C\rangle$ *and a compound label* L *with* $S_L \subseteq X$ *then* L *is a* ***solution*** *of* $\langle X, D, C\rangle$ *when* $\forall c \in C$ *:* $\textit{satisfies}(L, c)$.

To illustrate the definitions above, we return to the 8-queens example from the introduction chapter. The set of variables of the 8-queens problem is the set of the queens: $X = \{x_1, x_2, \ldots, x_8\}$. As there can not be more than one queen per column on the chessboard, each of the eight variables can take one of the eight rows as its value. Like in chess, the rows are labelled from 1 to 8. The domains of all variables are then defined as: $D_{x_1} = D_{x_2} = \ldots = D_{x_8} = \{1, 2, 3, 4, 5, 6, 7, 8\}$. The 8-queens problem has then two overall restrictions:

r_1: No two queens may be placed in the same row: $\forall i, j : i \neq j \Rightarrow x_i \neq x_j$ with $1 \leq i, j \leq 8$; and

r_2: No two queens may be placed diagonally from each other: $\forall i, j : i \neq j \Rightarrow |i-j| \neq |x_i - x_j|$ again with $1 \leq i, j \leq 8$.

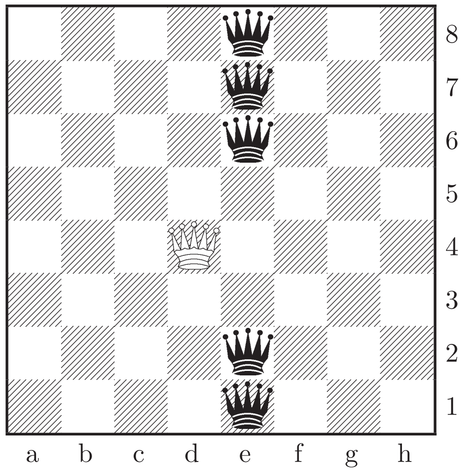

Figure 2.1: Construction of the $c_{x_4=4,x_5}$ constraint.

It is possible to combine these two restrictions into a single constraint. This constraint has the same variable set as the 8-queens problem itself. However, constructing this constraint would involve solving the 8-queens problem, as by definition it would contain all solutions of the problem. Instead we construct constraints per variable-pair, e.g., variables x_4 and x_5. We denote this constraint as c_{x_4,x_5}. We start the construction by placing a queen on row 4. Figure 2.1 shows this board. The black queens show the possible positions that queen x_5 may be placed on.

We define constraint $c_{x_4=4,x_5}$ as:

$$c_{x_4=4,x_5} = \{(\langle x_4,4\rangle\langle x_5,1\rangle),(\langle x_4,4\rangle\langle x_5,2\rangle),\\ (\langle x_4,4\rangle\langle x_5,6\rangle),(\langle x_4,4\rangle\langle x_5,7\rangle),(\langle x_4,4\rangle\langle x_5,8\rangle)\}.$$

The remaining combinations of the c_{x_4,x_5} constraint can be constructed by placing the (white) queen at the other 7 positions and merging the resultant compound label sets with the set already given. Repeating this for all $8\cdot(8-1)=56$ variable combinations of the 8-queens CSP fully defines the problem without actually solving it.

2.2 Binary Constraint Satisfaction Problems

Although the variable set S_c of constraint c can hold an arbitrary large number of variables, research in the constraint satisfaction problem usually limits the number of variables in S_c to two. A constraint with a variable set of only two variables is called a binary constraint.

Definition 2.15 (Binary Constraint)
A constraint c is a binary constraint if and only if the set of variables of the constraint S only contains two variables: $|S_c| = 2$.

A constraint satisfaction problem made up entirely out of binary constraints has an arity of two and is called a binary constraint satisfaction problem.

Definition 2.16 (Binary CSP)
*A **binary constraint satisfaction problem** is a CSP with only binary constraints.*

We will use BCSP to abbreviate binary constraint satisfaction problem.

Although the restriction to binary constraints appears to be a serious limitation to the constraint satisfaction problem, E. Tsang showed that every CSP can be transformed to an equivalent BCSP [83]. Two methods of translating constraint satisfaction problems of arbitrary arity to binary constraints satisfaction problems have been proposed: the dual graph translation by R. Dechter and J. Pearl ([23]) and the hidden variable translation by R. Dechter([22]).

In the dual graph translation, the constraints of the original problem become variables in the new representation. These variables represent the constraints and are referred to as *c-variables*. The domain of each c-variable is the set of compound labels of the original constraint. There is a binary constraint between two c-variables if and only if the original constraints share some variables. The binary constraints prohibit pairs of tuples in which shared variables receive different values.

In the hidden variable translation, the set of variables includes all of the variables of the original problem (their domains remain unchanged) plus a new set of "hidden" or *h-variables*. For each constraint in the original problem we add an h-variable. The domains of these variables consists of a unique identifier for every tuple in the constraint they represent. The new representation contains only binary constraints. They are constructed as follows. For every h-variable we impose a binary constraint between it and each of the variables in the set of variables of the original constraint. Say that x_h (the hidden variable) and x_i (the original variable) are thus constrained. Every value of x_h corresponds to a tuple of values for the variables in the set of variables of the original constraint and thus defines a unique value for x_i. Hence the binary constraint between x_h and x_i consists of a unique value for x_i for every value of x_h. Note that the constraint is not functional in the other direction as a value for x_i may be compatible with many values if x_i.

F. Bacchus and P. van Beek discussed both methods in [6]. There they posed the hypothesis that the choice of the transformation method has a large impact on the performance of the algorithm used to solve the resulting BCSPs. Because we can translate the CSP into the BCSP, from now on we will continue the discussion with BCSPs, although most of the discussion can also be generalised to CSPs.

2.3 Representing Constraint Satisfaction Problems

Sometimes it is useful to represent the constraint satisfaction problem in a way other than through the mathematical definitions above. There are two ways of doing this. The first uses matrices, the second graphs. Both ways of representing the constraint satisfaction problem have their advantages and disadvantages.

2.3.1 Matrix Representation

The matrix representation of a constraint satisfaction problem uses two types of matrices to define the problem. The first is called the constraint matrix and it is used to show which variables are in the variable set of each constraint.

Definition 2.17 (Constraint Matrix)
A ***constraint matrix*** R *of a binary constraint satisfaction problem* $\langle X, D, C\rangle$ *is a* $|C| \times |X|$ *matrix, such that:*

$$R(c, x) = \begin{cases} 1 & \textit{if } x \in S_c, \\ 0 & \textit{otherwise.} \end{cases}$$

with $c \in C$ *and* $x \in X$*.*

The second matrix type required by the matrix representation is called the conflict matrix. Each constraint in the constraint satisfaction problem has its own conflict matrix. The conflict matrix shows the compound labels in the constraint by a zero in the matrix. The compound labels not in the constraint are shown with a one in the matrix. As a matrix is a two dimensional representation, it is only used for binary constraints, although ternary constraints can be depicted using a cube.

Definition 2.18 (Conflict Matrix)
Given a binary constraint satisfaction problem $\langle X, D, C\rangle$*. A* ***conflict matrix*** $M_c^{x,y}$ *of a constraint* $c \in C$ *for variables* $x \in X$ *and* $y \in X$ *is then a* $|D_x| \times |D_y|$ *matrix, such that:*

$$M_c^{x,y}(p, q) = \begin{cases} 0 & \textit{if satisfies}((\langle x, d_p\rangle, \langle y, d_q\rangle), c), \\ 1 & \textit{otherwise.} \end{cases}$$

with $x \in S_c$*,* $y \in S_c$*,* $c \in C$*,* $1 \leq p \leq |D_x|$*,* $1 \leq q \leq |D_y|$*,* $d_p \in D_x$*, and* $d_q \in D_y$ *and the domains numbered.*

For an illustration of both matrices we turn again to the N-queens problem. In Table 2.1 the constraint matrix for the 4-queens problem is represented, in 2.2 the conflict matrix for constraint c_{x_2,x_3} is shown.

$\mathbf{c} \backslash \mathbf{x}$	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{x_4}$
$\mathbf{c_1}$	1	1	0	0
$\mathbf{c_2}$	1	0	1	0
$\mathbf{c_3}$	1	0	0	1
$\mathbf{c_4}$	0	1	1	0
$\mathbf{c_5}$	0	1	0	1
$\mathbf{c_6}$	0	0	1	1

Table 2.1: Constraint matrix of the 4-queens problem.

$\mathbf{x_2} \backslash \mathbf{x_3}$	**1**	**2**	**3**	**4**
1	1	1	0	0
2	1	1	1	0
3	0	1	1	1
4	0	0	1	1

Table 2.2: Conflict matrix of constraint c_{x_2,x_3} of the 4-queens problem.

The combination of the constraint matrix for a constraint satisfaction problem and the conflict matrices for the constraints in the constraint matrix fully defines the constraint satisfaction problem. However, this representation can be lengthy for large number of constraints. Because of its close relationship with arrays in computer languages however, it is commonly used in computer implementations of the constraint satisfaction problem.

2.3.2 Graph Representations

Two graph representations exist for CSPs . The first graph representation is called the constraint graph. It is used primarily to show which constraints are relevant to the variables of the CSP. In the graph, conflict matrices are used to show more restricted constraints from lesser ones. Because conflict matrices are defined for binary CSPs only, the constraint graph including the conflict matrices can only be used for binary CSP as well. Without the conflict matrices, the constraint graph can be defined for CSP with arbitrary arity by redefining the edges of the graph.

Definition 2.19 (Constraint Graph)
*A **constraint Graph** of a binary constraint satisfaction problem $\langle X, D, C\rangle$ is a graph $G_{\langle X,D,C\rangle} = \langle V, E\rangle$ where V is a set of vertices and E is a set of edges that are defined as follows: Every variable $x \in X$ is mapped to a vertex $v_x \in V$ and each constraint $c \in C$ for which $x \in S_c$, $y \in S_c$, and $x, y \in X$ is mapped to an edge such that $\langle v_x, v_y\rangle \in E$ if and only if $(\langle x, d\rangle, \langle y, d'\rangle) \ni c$ for some $d \in D_x$ and $d' \in D_y$. Every edge is assigned its constraint's conflict matrix $M_c^{x,y}$.*

The second graph representation of a BCSP is called the conflict graph. It is commonly used to show which variables are more restrictive than others. Each variable is represented as a set of vertices, one for each domain value of the variable. A vertex of one variable is connected by an edge to a vertex of another variable when the compound label representing these labels is not in the constraint relevant to the two variables. Because of the large number of vertices in the graph, the conflict graph is less informative about which constraints are relevant to which variables of the BCSP. Usually, the constraint graph and the conflict graph are used in conjunction with each other.

Definition 2.20 (Conflict Graph)
*A **conflict graph** of a binary constraint satisfaction problem* $\langle X, D, C\rangle$ *is a hypergraph* $\prod_{\langle X,D,C\rangle} = \langle V, E\rangle$ *where* V *is a set of vertices and* E *is a set of edges that are defined as follows: Every value* $d_i \in D_x$ *from every variable's (*$x \in X$*) domain is mapped to a vertex* $v_i \in V$ *and each compound label that occurs in a constraint* $c \in C$ *is mapped to an edge such that* $\langle v_x, v_y\rangle \in E$ *with* $x, y \in X$ *only if both* $x \in S_c$ *and* $y \in S_c$ *and* $(\langle x, v_x\rangle, \langle y, v_y\rangle) \ni c$*.*

For an illustration of the constraint graph and the conflict graph we return to the 4-queens problem. Figure 2.2 shows the constraint graph of the 4-queens problem and Figure 2.3 the conflict graph.

2.4 Constraint Satisfaction Problem Complexity

The difficulty of solving a problem class is expressed by the complexity of the best algorithm that was found for solving the problem-class. The complexity of an algorithm is the cost of using the algorithm to solve one of the problems. The cost is measured as the time units (computational complexity), the storage space (space complexity), or whatever units are relevant, needed by the algorithm to solve the problem. The study of the amount of computational effort that is needed in order to perform certain kinds of computation is the study of computational complexity. The complexity of an algorithm is measured by expressing the running time of an algorithm as a function of some measure of the amount of data that is needed to describe the problem to the algorithm.

The general rule is that if the running time of an algorithm is at most a polynomial function of the amount of data then the problem is *easy*, otherwise it is *hard*. Showing that a problem is easy is done by providing an algorithm that solves it in at most polynomial time. Showing that a problem is hard is not as easy as it has to be proved that no algorithm can be found that will solve it in polynomial time. The fact that a computational problem is hard does not mean that every instance of the problem has to be hard. The problem is hard because no algorithm can be devised for which a guarantee can be given that it will solve *all* instances in polynomial time.

A problem can be phrased to be a *decision problem* or an *optimisation problem*. A decision problem only provides a yes or no answer to a problem while a optimisation problem provides the optimal answer to a problem. Any optimisation problem can be

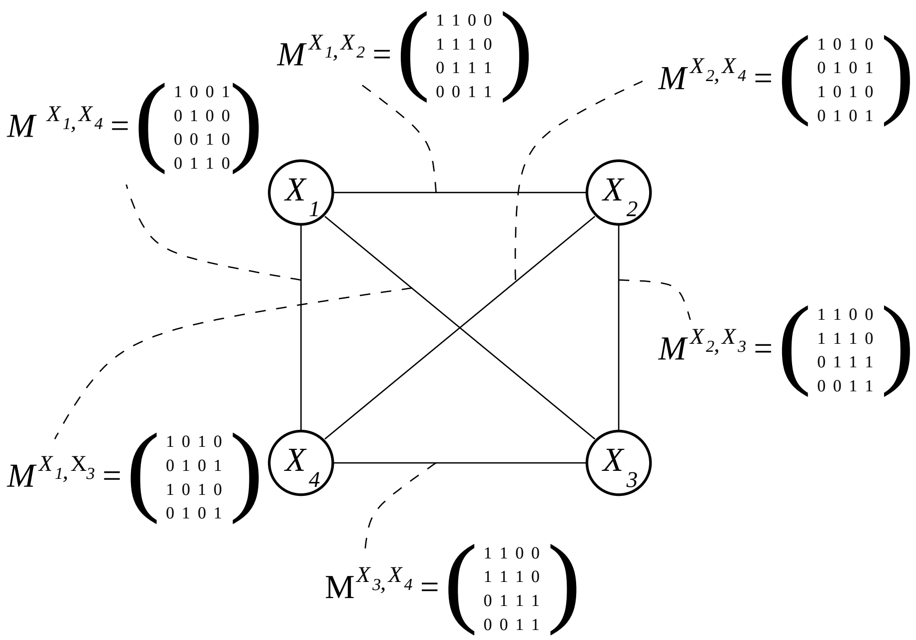

Figure 2.2: The constraint graph of the 4-queens problem.

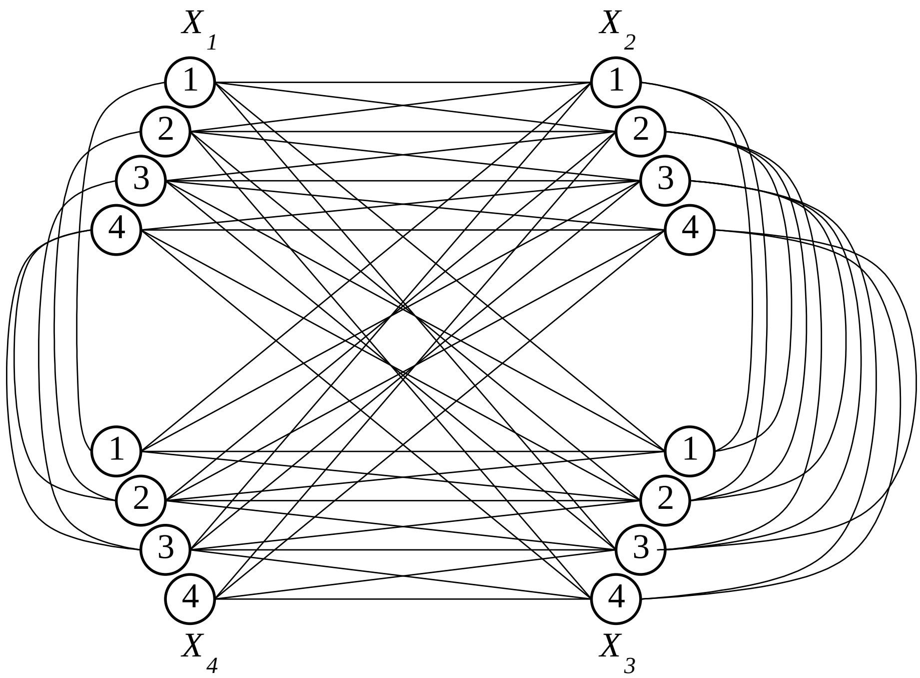

Figure 2.3: The conflict graph of the 4-queens problem.

solved by repeatedly solving a decision problem. We can think of a decision problem as asking if a given word (the input string) does or does not belong to a certain *language*. The language constitutes all words for which the decision problem would give a positive answer. A decision problem belongs to the class P when there is an algorithm A such that for every instance I of the problem, algorithm A will produce a solution in polynomial time as a function of the size of instance I. A decision problem Q belongs to NP if there is an algorithm A that: associates with each word of the language of Q a certificate $B(I)$ such that when the pair $(I, B(I))$ are input to algorithm A, it recognises that I belongs to Q; if I does not belong to Q then there is no $B(I)$ that will cause A to recognise I as a member of Q; operates in polynomial time. More briefly, P is the class of problems were it is easy to find a solution while NP is the class of problems for which it is easy to check the correctness of a solution. Note that $P \subset NP$ and that if decision problem $Q \in P$, membership in the language Q can be verified with an empty certificate. The question of whether or not $P = NP$ is perhaps the most important open question in the study of computational complexity.

Given decision problems Q and Q', Q' is *quickly reducible to* Q if whenever we are given an instance I' of Q' it can be converted to an instance I of Q in polynomial time, in such a way that both I and I' have the same answer. A decision problem is *NP-complete* if it belongs to NP and every problem in NP is quickly reducible to it. In 1971, S. Cook described NP-complete using the theory of *Turing Machines* [16]. A full description of the proof and of a Turing Machine is beyond the scope of this thesis. It suffices to say that the Turing Machine is used as a *checking* or *verifying* machine and that a Turing Machine used as such is called a *non-deterministic* machine. The name NP is derived from that name, standing for *non-deterministic polynomial*. In 1990, F. Rossi *et al.*, proved that the constraint satisfaction problem is in NP and that all NP-complete problems are quickly reducible to it [77].

As explained above, the complexity of a CSP is directly proportional to the size of the problem. The number of variables and the size of the domains of these variables define the size of the CSP and can be seen as complexity measures of an instance of the CSP. Two other complexity measures of a CSP instance can be defined, one being an average over yet another measure.

The first of these other complexity measures is called the density of a CSP.

Definition 2.21 (Density)
*The **density** of a binary constraint satisfaction problem is the ratio between the maximum number of constraints $\binom{|X|}{2}$ and the actual number of constraints $|C|$:*

$$p_1 = \frac{|C|}{\binom{|X|}{2}} \tag{2.1}$$

The second complexity measure is the average of one minus the ratio of the maximum number of compound labels to actual compound labels of all constraints in the BCSP. The parameter is called the *average tightness* of the BCSP, *tightness* itself is defined for a single constraint.

Definition 2.22 (Tightness)
The ***tightness*** *of a constraint c over variables $x, y \in X$ of a binary constraint satisfaction problem $\langle X, D, C\rangle$ is one minus the ratio between the maximum number of compound labels possible ($|D_x \times D_y|$) and the actual number of compound labels ($|c|$):*

$$p_2(c) = 1 - \frac{|c|}{|D_x \times D_y|}$$

Definition 2.23 (Average tightness)
The ***average tightness*** *of a constraint satisfaction problem $\langle X, D, C\rangle$ is the sum of the tightness over all constraints divided by the number of constraints:*

$$\overline{p_2} = \frac{\sum_{c \in C} p_2(c)}{|C|}$$

Unlike the number of variables and the domain sizes, the density and average tightness measures do not relate to the input size of the CSP. They are still complexity measures though as CSPs with more constraints (higher density) or less compound labels in their constraints (higher average tightness) are still harder to solve.

The four measures of the CSP allow for the definition of the parameter vector of a CSP, which is used as a short-hand description of a CSP. Using the parameter vector of a CSP assumes that the domain sizes of the variables are the same. A CSP with these domain sizes is said to have uniform domain sizes, or is called a uniform CSP.

Definition 2.24 (Parameter Vector of a BCSP)
The ***parameter vector of a BCSP*** $\langle X, D, C\rangle$ *is a quadruple $\langle n, m, p_1, \overline{p_2}\rangle$ of four parameters: the number of variables $n = |X|$, the domain size of each variable $m = |D_{x_1}| = |D_{x_2}| = \cdots = |D_{x_n}|$, the density p_1 and the average tightness $\overline{p_2}$.*

2.5 Generating Random Binary Constraint Satisfaction Problems

Finding more efficient algorithms to solve CSPs has been an important driving force behind the study of CSPs. The lack of a good set of problem instances to study was soon identified as a major obstacle in the research of CSPs. It was also soon realised that an algorithm that solved particular problem instances efficiently may have disappointing performance on other problem instances. This has led to research on how to produce sets of randomly created CSPs that qualify as a reasonable representation of the whole class. These sets can then be used to empirically research CSP solving algorithms.

Several models for randomly creating CSPs have been designed in the last two decades [69, 2, 56]. These models all use a similar parameter vector like the parameter vector of

Model	Constraints	Conflicts
Model A	probability model	probability model
Model B	ratio model	ratio model
Model C	probability model	ratio model
Model D	ratio model	probability model

Table 2.3: BCSP generator models.

a BCSP to control the size and complexity of the problems they generate. By analysing the performance of the algorithms on instances created with different parameter settings, the behaviour of the algorithms throughout the parameter space can be studied. A set of CSP instances for empirically testing the performance of an algorithm is called a *test-set*.

Generating CSP instances involves choosing which constraints to remove compound labels from and which compound labels to remove from these constraints. There are two methods for making these choices: the *ratio* method and the *probability* method. In the ratio method $p_1 \cdot \binom{n}{2}$ constraints are uniform randomly chosen and $1 - \overline{p_2} \cdot m^2$ compound labels are added to them. The ratio method is sometimes called the *uniform* method, as constraints and compound labels are chosen uniform randomly. The probability method considers every constraint and removes compound labels from it with probability p_1. The compound labels that are removed are chosen with probability $1 - \overline{p_2}$. Both methods share a method for choosing constraints and a method for removing compound labels from the chosen constraints. This makes for a total of four combinations of methods. In [69] and [56] these four combinations are designated as models A, B, C, and D. How the different methods combine into these models is shown in Table 2.3.

In [2], D. Achlioptas *et al.* showed that when the number of variables (n) of a randomly generated CSP is large, almost all instances created by models A, B, C, and D become unsolvable. The reason for this are the existence of *flawed variables*. A flawed variable is a variable for which all values in its domain violate a relevant constraint.

Definition 2.25 (Flawed variable)
Given a binary constraint satisfaction problem $\langle X, D, C\rangle$, a variable $x \in X$ is flawed if and only if:

$$\exists c \in C : \exists x, y \in S_c : \forall d \in D_x : \nexists d' \in D_y : \textit{satisfies}((\langle x, d\rangle\langle y, d'\rangle), c)$$

As the number of variables in CSP instances generated by models A to D increases and the complexity parameters remain the same, the probability of introducing a flawed variable increases, thereby also increasing the probability of generating an unsolvable CSP instances. This as a result of this model's two step approach for choosing constraints and compound labels. To overcome this unwanted behaviour, D. Achlioptas *et al.* introduced a new model, called model E, for generating CSPs. Model E generates CSP instances by choosing both constraints and compound labels at the same time.

Definition 2.26 (Model E)
The graph C_Π is a random n-partite graph with m nodes in each part. It is constructed by uniformly, independently and with repetitions selecting $(1 - p_e)\binom{n}{2}m^2$ edges out of the $\binom{n}{2}m^2$ possible ones.

Instead of using two complexity parameters; density (p_1) and average tightness ($\overline{p_2}$), model E uses a single complexity parameter: p_e. The parameter vector of model E is therefore defined as $\langle n, m, p_e \rangle$. Although parameter p_e could be said to control the average tightness of the generated CSP instances, it is not equal to the average tightness parameter of models A to D ($\overline{p_2}$) as the compound labels are added with repetition. There is a chance that some compound labels will be added more than once. The actual average tightness of a model E generated CSP instance will therefore be lower or at most equal to p_e.

An effect of generating CSP instances using a model E generator is that even with small values of p_e (e.g. $p_e < 0.05$), all possible constraints will be restrictive. E. MacIntyre *et al.* proposed a correction on model E in [56] by generating CSP instances in two phases: first generate a CSP instance using a model E generator and then choose $1 - (p_1\binom{n}{2})$ constraints uniform randomly and make them non-restrictive again. This method of generating CSP instances has become known as a model F generator. The parameter vector of a model F generator is $\langle n, m, p_1, p_e \rangle$. Note that the measured average tightness of a CSP instance generator by a model F generator is still lower than the p_e value used to generate the instance, as not only are compound labels chosen with repetition but some are added again when some constraints are made non-restrictive again in the second phase of the generation process. To generate a CSP instance by a model F generator with a specific average tightness value therefore necessitates experimental tweaking of the p_e parameter.

The pseudo-code for a model F CSP generator is given in algorithm 2.1. The operator $round$ in lines 22 and 34 is used to indicate that the result of the equation is rounded to the next natural number. The operator $random$ is used to indicate that a uniform random choice was made from the elements of a set, i.e., $random \in X$ (line 24 'selects' a variable from the set of variables uniform randomly.

Algorithm 2.1: The model F random binary CSP generator

```
1 funct modelF(n, m, p1, pe) ≡
2    X := ∅; D := ∅; C := ∅;
3    for x : 1 ≤ x ≤ n do
4        X := X ∪ {x};
5        Dx := ∅;
6        for dx : 1 ≤ dx ≤ m do
7            Dx := Dx ∪ {dx};
8        od
9        D := D ∪ {Dx};
```

```
    od
    for x : 1 ≤ x < n do
        for y : x < y ≤ n do
            c_{x,y} := ∅;
            for d_x : 1 ≤ d_x ≤ m do
                for d_y : 1 ≤ d_y ≤ m do
                    c_{x,y} := c_{x,y} ∪ {(⟨x, d_x⟩, ⟨y, d_y⟩)};
                od
            od
            C := C ∪ {c_{x,y}};
        od
    od
    conflicts := round(p_1 · p_e · n · (n − 1) · 0.5 · m · m);
    while conflicts > 0 do
        x := random ∈ X; y := random ∈ X;
        while x = y do
            y := random ∈ X;
        od
        if x > y
          then tmp := x; x := y; y := tmp; fi
        d_x := random ∈ D_x; d_y := random ∈ D_y;
        C := C ∩ {(⟨x, d_x⟩, ⟨y, d_y⟩)};
        conflicts − −;
    od
    constraints := |C| − round(p_1 · n · (n − 1) · 0.5);
    while constraints > 0 do
        x := random ∈ X; y := random ∈ X;
        while |c_{x,y}| = m · m do
            x := random ∈ X; y := random ∈ X;
        od
        for d_x : 1 ≤ d_x ≤ m do
            for d_y : 1 ≤ d_y ≤ m do
                c_{x,y} := c_{x,y} ∪ {(⟨x, d_x⟩, ⟨y, d_y⟩)};
            od
        od
        constraints − −;
    od
    exit(BCSP⟨X, D, C⟩)
end
```

Chapter 3

Classical Algorithms

In this chapter, two classical algorithms will be introduced: the *Chronological Backtracking Algorithm* and the *Forward Checking with Conflict-Directed Backjumping Algorithm*.

In the previous chapter, a solution of a constraint satisfaction problem was defined as a compound label over all variables of the problem such that all constraints are satisfied. However, finding such a solution is only one of four variants for solving a CSP:

1. finding a solution;
2. finding all solutions:
3. proving there is no solution;
4. find a compound label with the maximum number of variables.

All four variants are proven to be NP-complete, and are of the same order of difficulty. The first and second variants assume that the CSP is solvable. The third assumes that it is unsolvable and the fourth variant can be used for both solvable and unsolvable CSPs but reverts to the first variant if it is actually solvable.

An algorithm is *sound* when if it claims to have found a solution, that compound labels is in fact a solution to the problem. An algorithm is *complete* when, if the problem has a solution, the algorithm will be able to find it. For an algorithm to be both sound and complete it has to systematically check or discard all possible solutions of a problem. All considered classical algorithms are both sound and complete.

A sound and complete algorithm that can find a single solution (variant 1) can be used to solve a CSP according to the three remaining variants:

1. finding all solutions (variant 2) can be done by using the algorithm to find the first solution, removing it from the search space and iterating the process until no more solution can be found;

2. proving that no solution exists (variant 3) is done when the algorithm can not find a single solution;

3. finding the maximum compound label (variant 4) can be done by adjusting the algorithm so that it will always remember the maximum compound label found during the search. If a solution is found it will return the solution, and if no solution is found, it will return the stored maximum compound label.

Most research on the CSP focusses on algorithms that find a single solution.

3.1 The *Chronological Backtracking Algorithm*

The first sound and complete algorithm to find a solution of a CSP was proposed in 1965 by S. Golomb and L. Baumert [41], and is called the *Chronological Backtracking Algorithm* (*CBA*). The *CBA* uses the backtracking search method to find a single solution to the CSP. Based on this search method, a number of more efficient sound and complete algorithms have been developed. In [55], G. Kondrak and P. van Beek have placed these algorithms in a hierarchy based on the number of visited nodes and the number of consistency checks.

The basic backtracking search method is in effect a depth-first search of the problem search space. For the CSP, backtracking divides the problem into the sub-problem of labelling a single variable with a value that is consistent with earlier labellings. A label is consistent with earlier labellings when it satisfies all relevant constraints to earlier labelled variables. The backtracking search method for the CSP tries to label the variables in order. For each variable, all labels are tried. If no more labels can be tried for a variable, backtracking goes back (backtracks), to the previous variable. Backtracking terminates when a solution is found or when no more labels for the first variable can be tried.

The pseudo-code for the *Chronological Backtracking Algorithm* is given in algorithm 3.1.

Algorithm 3.1: The *Chronological Backtracking Algorithm*

$CSP\langle X, D, C\rangle$
funct $backtrack((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i) \equiv$
  if $i > |X|$ **then** **exit**$(TRUE)$ **fi**
  for $\forall d \in D_i$ **do**
    $v_i := d;$
    if $consistent((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i)$
      then
        if $backtrack((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i+1)$
          then **exit**$(TRUE)$ **fi**
    fi
  od
  exit$(FALSE)$

end

funct $consistent((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i) \equiv$
  for $\forall j : 1 \leq j < i \wedge j < |X|$ **do**
    $conflict\_checks++;$
    if $violates((\langle x_i, v_i\rangle, \langle x_j, v_j\rangle), c_{x_i,x_j})$
      then **exit**$(FALSE)$ **fi**
  od
  exit$(TRUE)$
end

3.2 The *Forward Checking with Conflict-Directed Backjumping Algorithm*

The *Forward Checking with Conflict-Directed Backjumping Algorithm* (abbreviated by *FCCDBA*) extends the *CBA* with two adaptations of the backtracking search method: *forward checking* [46], and *conflict-directed backjumping* [73]. Both extensions try to reduce the number of compound labels checked based on information already found during the search.

The *CBA* uses backtracking to check consistency from the currently considered label *back* to earlier labels. Forward checking in the *FCCDBA* reverses the process by a technique called *shrinking domains*. For each variable in the CSP, the domain is stored as a set of values, called the *domain set*. Like backtracking, forward checking tries to label the variables in order. The values used for labelling the variables are taken from their respective domain set. When forward checking labels a variable, it removes all values from the domain sets of the unlabelled variables that violate a relevant constraint with the current label. When the last value from a domain set is removed, the current label can never be part of a solution. The domain sets of the unlabelled variables are then restored and another value from the domain set of the current variable is tried. When no last variable from the domain set is removed, the next variable is labelled, and so on. When all values from the current domain set have been tried, forward checking backtracks to a previous variable. Forward checking terminates when a solution is found or when all values from the domain set of the first variable has been tried. In the latter case, the problem has no solutions.

The conflict-directed backjumping extension in the *FCCDBA* changes the way in which the algorithm backtracks to previous variables. Instead of backtracking to the previous variable, the *FCCDBA* uses information about which constraint was violated to determine which earlier variable to backtrack to. Each variable in the CSP is assigned a set of conflicting variables in the *FCCDBA* called the *conflict set of a variable*. Because forward checking is used, this set contains a set of as yet unlabelled variables that have failed a consistency check during forward checking. When all values from the domain set of the current variable have been tried, the algorithm backtracks to the earliest variable found in the conflict set. All conflict sets are then restored to the situation where the algorithm left off with that variable.

Both forward checking and conflict-directed backjumping use sets of either values or variables to reduce the number of compound labels that need to be checked for consistency. Forward checking uses domain sets for each variable to reduce the number of future labels that need to be checked. Conflict-directed backjumping uses conflict sets for each variable to backtrack to earlier variables further up the search tree. Both essentially increase space complexity for a decrease in computational complexity (see section 2.4 for description of space and computational complexity). The increase of space complexity is the product of the number of variables and the domain size of these variables for the forward checking extension. The increase of space complexity is cubic to the number of variables for the conflict-directed backjumping extension. For both extensions there is also a small increase of the computational complexity because these sets need to be maintained. The decrease in computational complexity is related to the complexity of the problem to solve. Constraint satisfaction problems with few constraints, or less restrictive constraints, benefit less from both extensions as the effect of domain shrinking is less and there is less chance of backjumping to an early variable. It is possible that the *CBA* outperforms the *FCCDBA* on easy constraint satisfaction problem.

The pseudo-code for *Forward Checking with Conflict-Directed Backjumping Algorithm* is shown in Algorithm 3.2.

Algorithm 3.2: The *Forward Checking with Conflict-Directed Backjumping Algorithm*

$CSP\langle X, D, C\rangle$
$conflictset[|X|][|X|] := -1;$
$checking[|X|][|X|] := FALSE;$
$domains[|X|][|D|] := -1;$
funct $FC\text{–}CBJ((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i) \equiv$
  if $i > |X|$ **then** **exit**$(TRUE)$ **fi**
  for $\forall d \in D_i$ **do**
    if $domains[i][d] = -1$
      then $v_i := d;\ end := FALSE;$
        for $\forall j : i < j \leq |X| \wedge end = FALSE$ **do**
          if $check\_forward((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i, j) = 0$
            then $end := TRUE$ **fi**
        od
        if $j = 0$
          then $j = FC\text{–}CBJ((\langle x_1, v_1\rangle, \ldots, \langle x_{|X|}, v_{|X|}\rangle), i + 1)$
            if $j \neq i$ **then** **exit**(j) **fi**
          else $union\_checking(i, j)$ **fi**
        $restore(i)$ **fi**
  od
  $j := 0;$
  for $\forall k : k < i \wedge k \leq |X|$ **do**
    if $conflictset[i][k] \neq -1$

```
            then j := k;  fi
   od
   for ∀l : j < l < i ∧ l ≤ |X| do
       if checking[l][i] = TRUE
          then j := l;  fi
   od
   union_checking(i, i);
   union_conflictset(j, i);
   for ∀m : j < m ≤ i ∧ m ≤ |X| step −1 do
       for n : n < m ∧ n ≤ |X| do
           conflictset[m][n] := −1;
       od
       restore(m);
   od
   if i ≠ 0 then restore(j);  fi
end

funct check_forward((⟨x_1, v_1⟩, . . . , ⟨x_|X|, v_|X|⟩), i, j)  ≡
   count := 0;  delete := 0;
   for ∀d ∈ D_j do
       if domains[j][d] = −1
          then count + +;  conflict_checks + +;
                 if violates((⟨x_i, v_i⟩, ⟨x_j, v_j⟩), c_{x_i,x_j})
                    then domains[j][d] := i;  delete + +;  fi fi
   od
   if delete > 0
     then checking[i][j] := TRUE;  fi
   exit(count − delete)
end

funct restore(i)  ≡
   for ∀j : j > i ∧ j ≤ |X| do
       if checking[i][j] = TRUE
          then checking[i][j] = FALSE;
                 for ∀d ∈ D_j do
                     if domains[j][d] = i
                        then domains[j][d] := −1;
                     fi
                 od
       fi
   od
end

funct union_checking(i, j)  ≡
   for ∀k : k < i ∧ k ≤ |X| do
       if conflictset[i][k] > −1 ∨ checking[k][j] = TRUE
```

```
            then conflictset[i][k] := 0;
             else conflictset[i][k] := −1;
        fi
    od
end

funct union_conflictsets(i, j) ≡
    for k : k < i ∧ k ≤ |X| do
        if conflictset[i][k] > −1 ∨ checking[j][k] = TRUE
            then conflictset[i][k] := 0;
             else conflictset[i][k] := −1;
        fi
        if conflictset[i][k] > 1 ∧ conflictset[k][k] < k
            then conflictset[i][i] = k;
        fi
    od
end
```

3.3 Performance Measures for Classical Algorithms

In the pseudo-code of the *CBA* (Algorithm 3.1) and the *FCCDBA* (Algorithm 3.2) the variable $conflict\_checks$ is increased every time a constraint is checked. Checking if a compound label is in the set of compound labels of a constraint is taken as the atomic step of the algorithm. These steps can be used to define performance measures. For classical algorithms one such step is called a conflict check:

Definition 3.1 (Conflict Check)
Testing if compound label L is in the set of compound labels of constraint c of a binary CSP is called a conflict check.

A classical algorithm is more efficient than another classical algorithm when it uses fewer conflict checks to find a solution. As such, the number of used conflict checks is a measure of the computational effort of an algorithm and it does not measure the space complexity of an algorithm. Both extensions of the *FCCDBA* increase the space complexity of the algorithm in order to reduce the number of conflict checks needed, e.g., the computational complexity. The increase in space complexity is linear in relation to the size of the problem of both extensions. As the increase in computation complexity is exponential relative to the size of the CSP, the increase in the space complexity of the *FCCDBA* is negligible. The same reasoning applies to the increase of computational complexity needed to handle the increase of space complexity for both extensions.

Chapter 4

Generating the Test-set

In this chapter a test-set of randomly generated constraint satisfaction problems will be created. This test-set will be used throughout the rest of the thesis for experimentation with evolutionary algorithms. Although the test-set is particularly useful for experimentation with evolutionary algorithms, it is equally useful for other non-deterministic algorithms as well.

The constraint satisfaction problem generators discussed in section 2.5 are non-deterministic algorithms. They all use random number sequences to make the choices necessary to generate a constraint satisfaction problem instance. A truly random sequence can only be generated by a truly random process. A truly random sequence can not be generated by a mathematical formula, for knowledge of the formula and sufficient numbers of the sequence already generated would enable someone to predict the next value with certainty. There are, however, formulae which *can* produce long sequences of numbers which satisfy many randomness criteria before they start to repeat. Such sequences are called *pseudo-random* and they are used by computers as a substitute for truly random number sequences. The most commonly used method for generating a pseudo-random number sequence of integers is based on a recurrence formula. Pseudo-random number generator using these formulae are called linear congruential generators. The sequence is initialised by a random-seed, a first value of the sequence, and the pseudo-random number generator will generate a different pseudo-random number sequence for each different random-seed value.

The constraint satisfaction problem generators discussed in section 2.5 use pseudo-random number sequences to make choices while generating a CSP instance. These choices include choosing which constraint to add or remove to the CSP instances and which compound label to add or remove from the constraints in the CSP instance. When different random-seeds are used, different choices are made, resulting in different CSP instances. This is independent of the complexity parameters used by the CSP generator.

Using different random-seeds, a CSP generator will produce different CSP instances. This feature is used to generate sets of different CSP instances for the same complexity parameters. Because different choices were made to generate the CSP instances in the set, the CSP instances in the set will show a variance in the complexity of the CSP instances. This will occur for example when the CSP generator chooses to remove a larger number of compound labels from constraints in the generation of one CSP instance than in generation of another CSP instance. When a large number of choices have to be made to generate a CSP instance, the probability of generating an outlier in complexity is small; approaching zero when the number of choices increases. It is impossible to predict the exact complexity of a randomly generated CSP instance.

This leads to the question of what a *representative* test-set of CSP instances is. A test-set is representative for a problem if it includes a large enough sample of instances of the problem such that it is an accurate description of the population of all problem instances. Obviously, a perfectly representative test-set includes all problem instances that are possible. For the CSP, even for small numbers of variables and small domain sizes, the population of all problem instances is so large that experimenting with such a test-set would be prohibitively expensive. In this chapter we provide a method of selecting a small number of problem instances so that experimentation with the test-set can be performed in a reasonable time. There is however another matter to consider. The test-set is intended for use with evolutionary algorithms and evolutionary algorithms are incomplete. Practically, this means that problem instances that are unsolvable will take the maximum allowed amount of effort of the evolutionary algorithm to solve. As such, it makes no sense to include them as no additional information about the effectiveness and efficiency of the algorithm can be gained from including them in the test-set. Excluding unsolvable problem instances means that the method for selecting the problem instances for the test-set has to take into account that the test-set is no longer representative of the population of all possible CSP instances but that it is representative of the population of all solvable CSP instances. Because of this, we will call our test-set an appropriate test-set instead of a representative test-set.

4.1 Test-set Parameters

The CSP instances in the test-set are generated using the model F CSP generator. The parameter vector of the model F CSP generator includes four parameters: n for the number of variables, m for the uniform domain size, p_1 for the constraint density and p_e as an average tightness parameter. Because the model F generator chooses the compound labels not in the CSP instance with repetitions and a number of constraints will be removed as well, the p_e parameter has to be set higher than the desired average tightness of the CSP instance. The generator is therefore implemented in such a way that it will approximate the desired average tightness ($\overline{p_2}$) by increasing the p_e parameter in a stepwise fashion. In the following discussion therefore, the approximated average tightness $\overline{p_2}$ will be used, instead of the actual p_e values.

The hardness of a CSP instance is measured by the number of solutions it has. Using a sound and complete algorithm, the number of solutions and thus the exact hardness of a CSP instance can be calculated. We used the *Chronological Backtracking Algorithm* to do this. In [81], Smith provided a formula for the number of solutions of a CSP instance based on the four complexity parameters that were used to generate it:

$$E(\text{number of solutions}) = x_e = m^n(1 - \overline{p_2})^{\binom{n}{2}p_1} \tag{4.1}$$

The formula only holds for binary CSPs with a uniform domain size. We will denote the number of solutions by x_e.

In [15], the authors demonstrate that all NP-complete problems go through a *phase-transition*. All NP-complete problems, including the CSP, have a so-called *transition point* which marks the spot in the parameter space where problems go from having many solvable problem instances to having almost no solvable problem instances. For many NP-complete problems, this transition point has been located empirically ([15, 65]). For CSPs, Smith predicted that it would occur around problem instances with only one solution ([81]), assuming that this solution will be hard to find among all other possible candidate solutions. When this assumption is combined with equation 4.1 it leads to:

$$m^n(1 - \overline{p_2})^{\binom{n}{2}p_1} = 1 \tag{4.2}$$

The transition point of a CSP occurs for those combinations in the parameter space where there is a 50% chance of generating a solvable CSP and consequently a 50% chance of generating an unsolvable CSP ([65, 82, 20]). Usually the number of variables and their uniform domain sizes are fixed and the density and average tightness are varied, so that there is not a transition *point*, but a *transition line* through the density and average tightness parameter space. As binary CSPs have finite discrete domains, the phase transition does not occur abruptly, but over a wider area in the parameter space. This area is called the *mushy region*.

In Figure 4.1, the transition lines for combinations of n and m are shown in the parameter space bound by density (p_1) and average tightness ($\overline{p_2}$). The x-axis shows the density, the y-axis the tightness. Eight (n, m)-combinations are shown from $(n, m) = (5, 5)$ to $(40, 40)$ in increments of 5 for both n and m.

A transition line divides the parameter space of the CSP into three regions:

1. The mushy region, already described;

2. The solvable region, in Figure 4.1 below the mushy region. CSP instances generated with the parameters in this region are almost exclusively solvable; and

3. The unsolvable region, in Figure 4.1 above the mushy region. CSP instances generated with parameters in this region are almost exclusively unsolvable.

In Figure 4.1, we see that for combinations of larger n and m, the solvable region decreases in size, while for combinations of smaller n and m the solvable region increases in size.

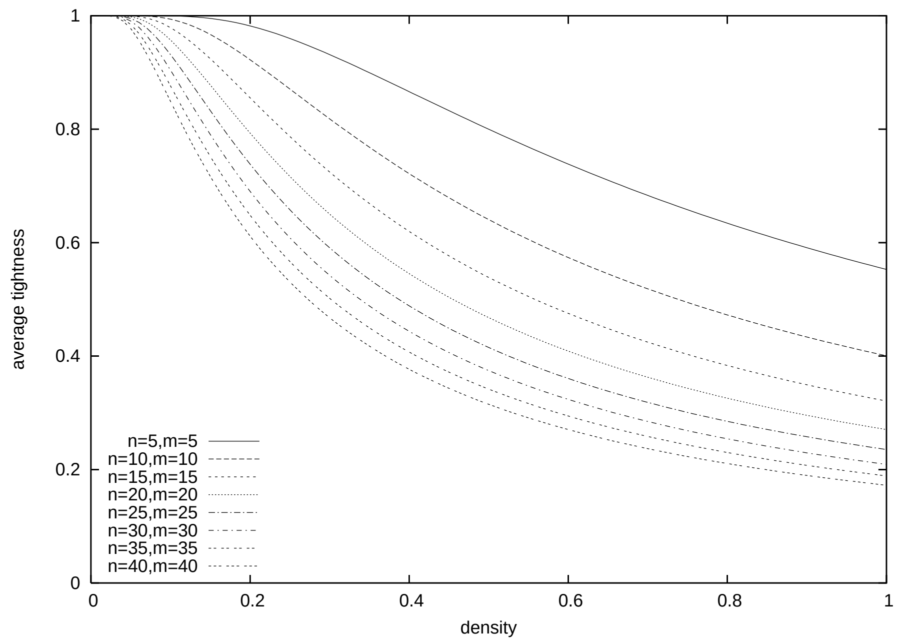

Figure 4.1: Transition lines for combinations of n and m found using Smith's formula.

As with all incomplete algorithms, evolutionary algorithms are, in general, unable to determine whether or not a problem is unsolvable. When they are used to solve an unsolvable problem they will continue trying to solve it until the maximum number of search steps allowed has been reached. The inclusion of unsolvable CSP instances in the test-set will only increase experimental effort without providing more insight into the performance of the algorithms. As such, we have decided not to include them.

Given the information above, we make the following considerations for the choice of the number of variables (n) and the uniform domain size (m) of the CSP instances in the test-set. The considerations are listed in order of importance.

1. The n and m parameters should be large enough to make solving the CSPs non-trivial.

2. The n and m parameters should be small enough to reduce the amount of experimental effort.

3. The n and m parameters should be chosen in such a way that the solvable region is large enough to include enough density-tightness combinations for adequate experimentation.

Obviously, considerations 1 and 2 are conflicting. As a practical compromise we have chosen to generate CSP instances with 10 variables and a uniform domain size of 10

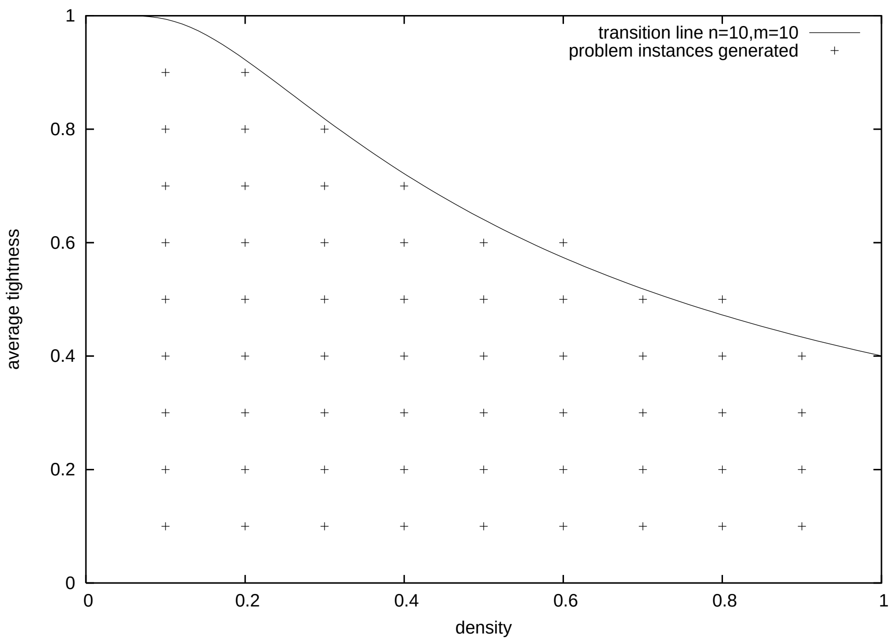

Figure 4.2: Overview of the parameter setup of the test-set with $n = 10$ and $m = 10$.

for our test-set. These parameter values will produce CSP instances with a maximum of $\binom{10}{2} = 45$ constraints and a maximum of 10^{10} possible candidate solutions to search through. We found that these CSP instances were by no means trivial to solve. The experimental effort needed to solve one of these CSP instances however is not prohibitive for a thorough investigation. On an average computer the *Chronological Backtracking Algorithm* needs less than a second to find a single solution and about a minute or two to find all solutions when the CSP instance lies within the mushy region.

Consideration 3 is related to the way CSP test-sets are commonly organised. Usually, a CSP test-set is constructed by generating a set of CSP instances for a number of density and tightness combinations with fixed parameters for the number of variables and the uniform domain size of these variables. The density and tightness combinations are chosen so that they form a grid-like pattern over the density-tightness parameter space. We used the following values for both density and tightness: $\{0.1, 0.2, \ldots, 0.9\}$. These values produce a grid-like pattern of 81 density-tightness combinations. When 10 variables with a domain size of 10 are used, 59 grid points lie in the solvable and mushy region of the density-tightness parameter space.

Figure 4.2 shows a graphical depiction of the parameter setup of the test-set. The line signifies the transition line found using Smith's formula for $n = 10$ and $m = 10$, copied from Figure 4.1. The sets of CSP instances for the different density-tightness combinations that are included in the test-set are shown as points in the solvable and mushy region. 59 sets will be generated. The mushy region is identified as the following list of density-tightness combinations: $(p_1, \overline{p_2}) \in \{(0.1, 0.9)$, $(0.2, 0.9)$, $(0.3, 0.8)$, $(0.4, 0.7)$, $(0.5, 0.6)$, $(0.6, 0.6)$, $(0.7, 0.5)$, $(0.8, 0.5)$, $(0.9, 0.4)\}$.

The most important sets of CSP instances in the test-set are found in the mushy region. The CSP instances in these sets will be the hardest to solve. Compared to the hardness of the CSP instances in these sets the hardness of the other CSP instances in the test-set is low. Algorithms solving CSP instances outside the mushy region should have little difficulty finding a solution. The CSP instances in the solvable region are therefore generated only for comparison with earlier research. In the rest of this chapter we will therefore focus mostly on making the sets of CSP instances in the mushy region as accurate as possible. The other CSP instances will be generated by simply using different random-seeds, without further analysis. For each density-tightness combination in the test-set, 25 instances will be generated.

4.2 Constructing a Test-set in 4 steps

In the previous section we decided to construct a test-set with CSP instances with 10 variables and a uniform domain size of 10. The CSP instances will be generated for 59 density-tightness combinations of which 9 lay in the mushy region of the density-tightness parameter space. The set of CSP instances with a specific density-tightness combination we will call the *sample* for that density-tightness combination. Each sample consists of 25 CSP instances.

Now that we have set the parameters for the CSP instances to be generated we can generate an appropriate test-set. We propose that the following properties for the CSP instances in each sample are necessary for constructing an appropriate test-set:

1. All CSP instances in each sample should be solvable;
2. The average number of solutions of the CSP instances in all samples should approximate the number of solutions calculated by using Smith's formula.
3. The variance in the number of solutions should be minimal over all CSP instances in each sample.

Formula 4.1 is defined for sets of both solvable and unsolvable instances. Because of requirement 1 the samples in the test-set contain only solvable instances. Therefore further analysis is necessary to see if we can use Smith's formula for samples of only solvable instances. This analysis is also necessary to see if Smith's formula is an accurate approximation of the number of solutions for CSP instances generated with a model F CSP generator. We will, therefore, first analyse samples of both solvable and unsolvable CSP instances and adjust the estimated number of solutions when necessary. The adjusted number of solutions will then be used to sub-sample a sample of only solvable instances in order to minimise the variance of the number of solutions. This final sub-sample should then have the properties mentioned above.

The method used to construct the test-set then consists of four steps:

Step 1: parameter adjustment Check if the values used for the CSP generated are equal to the parameters that should be used in Smith's formula. Because the CSP generator will choose discrete numbers of constraints and compound labels and Smith's formula uses real numbers, it is safe to assume that there will be a difference between the two parameter vectors used. The different parameters will produce different calculated number of solutions and an adjustment will have to be made for this. We will use x'_e to indicate the adjusted number of solutions.

Step 2: sample sizing The test-set construction method described below depends for a large part on statistical analysis. For statistical analysis to be accurate, a large sample of CSP instances is necessary. In this step we generate a large sample of CSP instances for each density-tightness combination in the mushy region. For each CSP instance in the sample the number of solutions is calculated using a classical algorithm. The average number of solutions of the sample, denoted by $\overline{x}$, is then compared to the adjusted number of solutions found in the first step. If the difference between $\overline{x}$ and x'_e is significant, this could be the result of not having generated enough CSP instances for the samples. We therefore generate more CSP instances until either the difference between $\overline{x}$ and x'_e becomes insignificant or a maximum practical sample size of 1000 CSP instances has been reached. If the difference between $\overline{x}$ and x'_e is still significant, continue with Step 3, otherwise continue with Step 4.

Step 3: formula correction Because we generated samples with a large number of CSP instances, we can assume that the difference between $\overline{x}$ and x'_e is not due to having too small a sample. The difference is most likely caused by Smith's formula calculating an inaccurate number of solutions. We therefore have to analyse the relationship between $\overline{x}$ and x'_e to see if the over- or under-estimation is systematic. If it is, we can correct x'_e for this, resulting in the corrected number of solutions, denoted by x''_e. We then have to analyse the difference between $\overline{x}$ and x''_e to see if it is significant. If it is, we have to consider another correction, if it is not we continue with Step 4.

Step 4: CSP instance selection With $\overline{x}$ approximately equal to either x'_e or x''_e, we will use it to sub-sample a sample of only solvable CSP instances. The single criterion for the sub-sampling is to minimise the variance of the hardness of the sub-sample. We do this by generating new samples for each density-tightness combination in the mushy region consisting of only solvable instances. The new samples are equal in size to the samples generated in Step 1. For each CSP instance in the sample, the number of solutions is calculated using a sound and complete algorithm. The CSP instances in these samples are ordered according to the difference between the calculated number of solutions of the CSP instances and either x'_e or x''_e, depending on whether step 3 was necessary. The sub-samples in the mushy region consist of the 25 instances with the smallest difference.

In Steps 2 and 3, the difference between the average number of solutions of the sample and the estimated number of solutions is used as a test. This involves a statistical analysis using the following hypothesis:

Hypothesis 4.1
In the mushy region the average number of solutions ($\overline{x}$) of a given sample is equal to the estimated number of solutions (x_e):

$$H_0 : \overline{x} = x_e$$
$$H_a : \overline{x} \neq x_e$$

In Steps 2 and 3, the adjusted number of solutions (x'_e) or the corrected number of solutions (x''_e) will replace x_e in the hypothesis.

The null-hypothesis (H_0) is rejected when the 5% margin of error between $\overline{x}$ and x_e (or x'_e, x''_e) is exceeded. For the hypothesis test we calculate the 95% confidence interval of the samples. If the number of solutions (x_e, x'_e, or x''_e) lies outside the confidence interval, the null-hypothesis is rejected. The confidence interval of a sample of size N of a population having unknown mean μ with known standard deviation σ is calculated as follows:

$$\overline{x} \pm z^* \frac{\sigma}{\sqrt{N}} \tag{4.3}$$

where z^* is the value on the standard normal curve with area C between $-z^*$ and z^*. C is exact when the population distribution is normal and is approximately correct for large N in other cases. C denotes the confidence interval.

The calculation of the confidence level assumes that the distribution of the sample points is normal. This we can not assume for the samples generated here. The *central limit theorem* states that when we draw a simple random sample from any population with finite standard deviation, the sampling distribution of the sample mean is approximately normal. The size of the sample needed to get a close approximation of the mean depends on the population distribution. We implement this by splitting the sample into 25 equal parts and calculating the mean for each of these parts. According to the central limit theorem, the distribution over these means approximates a normal distribution. The confidence interval of hypothesis 4.1 is calculated over these 25 means.

4.2.1 Step 1: Parameter Adjustment

Smith's formula uses four parameters to calculate the number of solutions: n for the number of variables, m for the uniform domain size, p_1 for density, and $\overline{p_2}$ for average tightness. The parameters are the same as the parameters in the parameter vector of the model F CSP generator. The last parameter of the model F CSP generator is different but as we approximate $\overline{p_1}$ by a stepwise increase of p_e, we can use $\overline{p_1}$ instead. Smith's formula uses the four parameters to exactly calculate the number of solutions meaning that it will take fractional constraints and compound labels into account. The model

$\mathbf{n}$	$\mathbf{m}$	$\mathbf{p_1'}$	$\mathbf{\overline{p_2}}$	$\mathbf{x_e'}$
10	10	0.1111	0.9	100000
10	10	0.2	0.9	10
10	10	0.3111	0.8	1.638
10	10	0.4	0.7	3.874
10	10	0.5111	0.6	7.037
10	10	0.6	0.6	0.180
10	10	0.7111	0.5	2.328
10	10	0.8	0.5	0.146
10	10	0.9111	0.4	8.020

Table 4.1: x_e' calculated using the actual density (p_1') values.

F CSP generator can not do this, the number of generated constraints and the number of generated compound labels is by definition integer. The model F CSP generator does this by rounding the number of constraints and the number of compound labels to the next nearest integer number. When the number of solutions of the generated CSP instances is calculated this behaviour will introduce a difference between calculated number of solutions by Smith's formula and the number of solutions. We will compensate for this difference by adjusting the density and the average tightness of the generated CSP instances and use these parameters to calculate the number of solutions by Smith's formula. We will use p_1' and $\overline{p_2}'$ to denote the adjusted density and average tightness.

The adjusted density of a binary CSP instance can be calculated by:

$$p_1' = \frac{\|\binom{n}{2} \cdot p_1\|}{\binom{n}{2}} \tag{4.4}$$

where n is the number of variables of the CSP instance to be generated and $\|\cdot\|$ is used to denote rounding to the next discrete number. The CSP instances to be generated for the test-set have 10 variables ($n = 10$) so they can have a maximum of $\binom{10}{2} = 45$ constraints. For density values $p_1 \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, the actual density values of the CSP instances are $p_1' \in \{0.1111\ldots, 0.2, 0.3111\ldots, 0.4, 0.5111\ldots, 0.6, 0.7111\ldots, 0.8, 0.9111\ldots\}$. The rounding difference between p_1 and p_1' is therefore $0.0111\ldots$ for density values $p_1 \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$.

Because of the larger number of conflicts to be generated for a CSP instance, the rounding difference between $\overline{p_2}$ and $\overline{p_2}'$ is usually negligible. The adjusted average tightness of a binary CSP instance can be calculated by:

$$\overline{p_2}' = \frac{\|\binom{n}{2} \cdot p_1' \cdot m^2 \cdot \overline{p_2}\|}{\binom{n}{2} \cdot p_1' \cdot m^2} \tag{4.5}$$

where n is the number of variables and m is the uniform domain size and $\|\cdot\|$ is again

$\mathbf{p_1'}$	$\overline{\mathbf{p_2}}$	$\mathbf{x_e'}$	$\overline{\mathbf{x}}$	$\mathbf{s}$	$\mathbf{CI_{95\%}}$
0.1111	0.9	100000	78127	10217	(73910,82345)
0.2	0.9	10	6.743	6.482	(4.067,9.419)
0.3111	0.8	1.638	1.114	0.678	(0.834,1.394)
0.4	0.7	3.874	3.015	1.059	(2.578,3.452)
0.5111	0.6	7.037	5.798	1.511	(5.174,6.422)
0.6	0.6	0.180	0.117	0.092	(0.079,0.155)
0.7111	0.5	2.328	1.937	0.668	(1.661,2.213)
0.8	0.5	**0.146**	0.118	0.076	(0.087,0.149)
0.9111	0.4	8.020	7.269	1.310	(6.728,7.810)

Table 4.2: Statistical analysis of $\overline{x}$ and x_e' for the samples of 1000 CSP instances in the mushy region.

used to denote rounding to the next integer. A maximum of $m^2 = 100$ conflicts can be generated for each constraint. Using the actual density values calculated above paired with the average tightness values $\overline{p_2} \in \{0.9, 0.9, 0.8, 0.7, 0.6, 0.6, 0.5, 0.4\}$ (in order), the actual average tightness values for the CSP instances in the mushy region can be calculated. No rounding difference between the expected average tightness values and the actual average tightness values was found: $\overline{p_2} = \overline{p_2}'$.

The difference between p_1 and p_1' results in different calculated number of solutions (x_e'). Table 4.1 shows the number of solutions calculated we p_1' is used.

4.2.2 Step 2: Sample Sizing

The statistical analysis in the following steps needs a large enough sample to be accurate. A sample of CSP instances is large enough when the null hypothesis of hypothesis 4.1 is valid. If the null hypothesis of hypothesis 4.1 is valid for a sample size smaller or equal to the maximum sample size (1000 CSP instances) we continue with Step 4, if not, further modifications of the estimated number of solutions is necessary (Step 3). The maximum sample size of 1000 CSP instances was chosen to place a limit on the effort needed to generate the sample and calculate the number of solutions for each instance in the sample. The number of solutions of each instance in the sample is calculated using the *Chronological Backtracking Algorithm*.

At first a sample of 100 CSP instances was generated for each density-tightness combination in the mushy region. The exact number of solutions for each CSP instance was then determined by the *CBA* algorithm. The samples were then uniform randomly divided into 25 sub-samples of 4 instances each. The average number of solutions was calculated over the average number of solutions of each sub-sample. As the adjusted number of solutions did not fall within the 95% confidence interval of the average number of solutions of the sub-samples, H_0 of hypothesis 4.1 had to be rejected. Next we tried samples with 200, 400 and finally 1000 instances. Again, all samples were divided into 25 equal sub-samples. The same hypothesis test was applied to all samples.

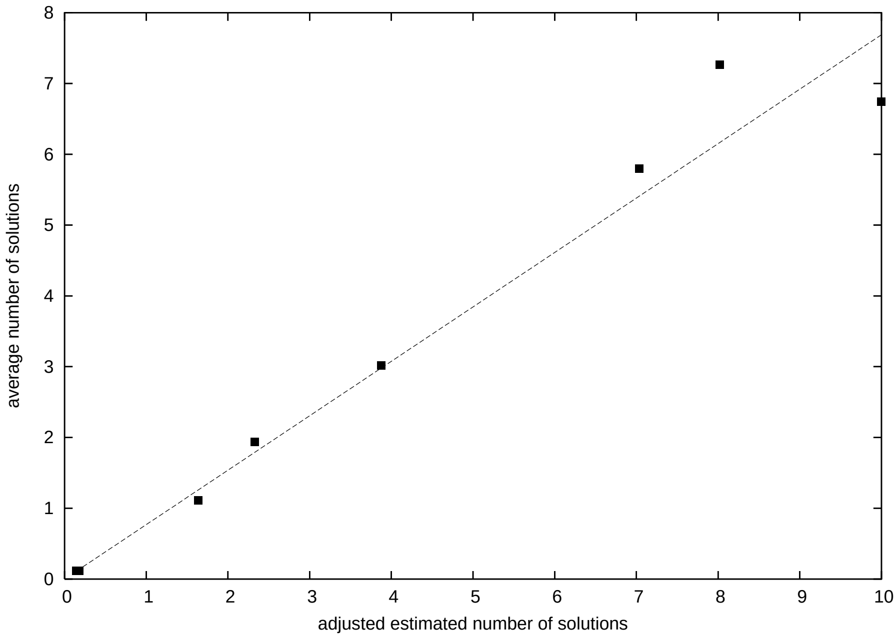

Figure 4.3: Scatter plot of x'_e and $\overline{x}$, excluding $(p_1, \overline{p_2}) = (0.1, 0.9)$.

Table 4.2 shows the statistical analysis of the samples with 1000 CSP instances. The first two columns show the actual density (p'_1) and the tightness ($\overline{p_2}$) values of the instances in the samples. The x'_e column shows the estimated number of solutions for these density-tightness combinations found in Step 1. The $\overline{x}$ column shows the mean of means of the sub-samples and the s column shows the standard deviation over these means. Column $CI^{95\%}$ shows the 95% confidence interval of the samples. Only for $(p'_1, \overline{p_2})$-combination $(0.8, 0.5)$ does the adjusted estimated number of solutions fall within the 95% confidence interval. For all other combinations hypothesis 4.1 has to be rejected. The estimated number of solutions need to modified further, we have to continue with Step 3.

4.2.3 Step 3: Formula Correction

In Step 2, we found that the adjusted number of solutions for all but one sample did not fall within the 95% confidence interval and that for these samples the null hypothesis of hypothesis 4.1 had to be rejected. We take this as an indication of the fact that the difference between the adjusted number of solutions found by Smith's formula and the average number of solutions calculated by a classical algorithm is not caused by having samples of insufficient size. We hypothesise that it is the result of a systematic error in Smith's formula. By analysing the relationship between the adjusted number of solutions and the number of solutions calculated by the *Chronological Backtracking*

$\mathbf{p_1'}$	$\overline{\mathbf{p_2}}$	$\mathbf{x_e''}$	$\overline{\mathbf{x}}$	$\mathbf{s}$	$\mathbf{CI_{95\%}}$
0.1111	0.9	**76888**	78127	10217	(73910,82345)
0.2	0.9	**7.6888**	6.743	6.482	(4.067,9.419)
0.3111	0.8	**1.2594**	1.114	0.678	(0.834,1.394)
0.4	0.7	**2.9786**	3.015	1.059	(2.578,3.452)
0.5111	0.6	**5.4106**	5.798	1.511	(5.174,6.422)
0.6	0.6	**0.1384**	0.117	0.092	(0.079,0.155)
0.7111	0.5	**1.7900**	1.937	0.668	(1.661,2.213)
0.8	0.5	**0.1123**	0.118	0.076	(0.087,0.149)
0.9111	0.4	**6.1664**	7.269	1.310	(6.728,7.810)

Table 4.3: Statistical analysis of $\overline{x}$ and x_e'' for the samples in the mushy region.

Algorithm, we can correct the adjusted number of solution for this difference. On inspection of the adjusted number of solutions we decided to treat $(p_1', \overline{p_2}) = (0.111, 0.9)$ as an outlier because the value for that sample is so large compared to the other values. Figure 4.3 shows the relation between the adjusted number of solutions and the calculated average number of solutions as a scatter plot. Along the x-axis the adjusted number of solutions (x_e') is shown, along the y-axis the calculated average number of solutions is shown.

The points in Figure 4.3 lie along a straight line. This indicates a linear relationship. The strength of the relationship is calculated by the correlation coefficient r. The closer the correlation coefficient is to 1.0, the stronger the relation. The correlation coefficient is calculated by:

$$r = \frac{1}{n-1} \sum \left(\frac{\overline{x}_i - \overline{\overline{x}}}{s_{\overline{x}}}\right)\left(\frac{x_{e,i}' - \overline{x_e'}}{s_{x_e'}}\right) \tag{4.6}$$

where $\overline{x}_i$ stands for the i-th value of $\overline{x}$, $\overline{\overline{x}}$ for the average over all values of $\overline{x}_i$, $s_{\overline{x}}$ for the standard deviation over all values of $\overline{x}_i$, $x_{e,i}'$ for the i-th value of x_e', $\overline{x_e'}$ for the average over all values of x_e', and $s_{x_e'}$ for the standard deviation over all values of x_e'. The correlation coefficient for x_e' and $\overline{x}$ is $r = 0.98121$, indicating a strong relationship. When $(p_1', \overline{p_2}) = (0.1111, 0.9)$ is included, the correlation coefficient is 1.0, but this is probably inaccurate.

The linear relationship between x_e' and $\overline{x}$ can be expressed by:

$$x_e' = \alpha \cdot \overline{x} + \beta \tag{4.7}$$

where α is the slope of the line through the data points and β is the intercept, the value of x_e' when $\overline{x} = 0$. Here the intercept is $\beta = 0$. The slope of the line through the points in Figure 4.3 can be calculated by:

$$\alpha = r \cdot \frac{s_{x_e'}}{s_{\overline{x}}} \tag{4.8}$$

$\mathbf{p_1'}$	$\overline{\mathbf{p_2}}$	$\overline{\mathbf{x}}_{\text{subsample}}$	$\mathbf{s}_{\text{subsample}}$
0.1111	0.9	77340	1289.3
0.2	0.9	8	0
0.3111	0.8	1	0
0.4	0.7	3	0
0.5111	0.6	5	0
0.6	0.6	1	0
0.7111	0.5	2	0
0.8	0.5	1	0
0.9111	0.4	6	0

Table 4.4: Mean and standard deviation of the sub-samples in the mushy region.

where r stands for the correlation coefficient, $s_{x'_e}$ for the standard deviation of x'_e and $s_{\overline{x}}$ for the standard deviation of $\overline{x}$, the latter two calculated over the values from the scatter plot. The slope of the straight line through the data points in the scatter plot is $\alpha = 0.76888$, the relationship found is then $\overline{x} = 0.76888 \cdot x'_e$. This relationship is shown in Figure 4.3 by the dotted line.

We use this relationship to correct the adjusted number of solutions a second time, by introducing a correction factor. The correction of the adjusted number of solutions is denoted by x''_e. Table 4.3 shows the statistical analysis of the samples using x''_e. The other columns of the table are copied from Table 4.2. The corrected number of solutions all fall inside the confidence interval of their respective samples. The null hypothesis of hypothesis 4.1 is valid when the corrected number of solutions is used. No further correction of the number of solutions is necessary: we can continue with Step 4.

4.2.4 Step 4: CSP Instance Selection

With either x'_e or x''_e, Step 4 is used to finish constructing the test-set. We first generated 1000 new samples of solvable CSP instances for each density-tightness combination in the density-tightness parameter space. The *FCCDBA* was used to calculate if the CSP instance is solvable. If not, another CSP instance was generated until a solvable one was generated. Using the *Chronological Backtracking Algorithm* we calculated the number of solutions for each CSP instance in these samples. The samples were then ordered according to the difference of the calculated number of solutions and either x'_e or x''_e. From each sample the 25 CSP instances with the least difference was selected for the test-set. In Table 4.4 the average number of solutions and the standard deviation for the selected instances in the mushy region are shown.

The nine sub-samples in the mushy region added to the uniform randomly generated samples from the solvable region form the test-set that will be used throughout the rest of the thesis.

Chapter 5

Iterated Local-Search and Evolutionary Algorithms

Evolutionary algorithms belong to a group of algorithms called Iterated Local-Search algorithms (ILS). The Iterated Local-Search meta-heuristic can be described in a nutshell as follows: a sequence of candidate solutions is built iteratively by an embedded heuristic, leading to better candidate solutions than if repeated random trials of that heuristic were used. This simple idea ([12]) has a long history and has lead to many differently named algorithms: iterated descent [11, 10], large-step Markov chains [61], iterated Lin-Kernighan [53], chained local optimisation [60], or combinations of these [3]. The historical development of iterated local-search algorithms can be found in [54].

An algorithm is considered a local-search algorithm when there is a single chain of candidate solutions that is followed, and the search for better candidate solutions occurs in a reduced space defined by the output of an embedded heuristic. In practice, local-search has been the most frequently used embedded heuristic, but in fact, any optimiser can be used, be-it deterministic or not. Although the description limits the algorithm to following only a single chain of candidate solutions, often more than one chain is followed concurrently. These algorithms are still considered to be ILS algorithms although they are also called concurrent ILS algorithms or population-based ILS algorithms.

In essence, an ILS algorithm consists of two parts: a move operator containing the embedded heuristic and a selection operator. The move operator is used to search through the search space of the problem. The selection operator is used to direct the search by selecting candidate solutions for the next iteration of the algorithm. The basic pseudo-code of an ILS algorithm is shown in algorithm 5.1.

Algorithm 5.1: The *Iterated Local Search Algorithm*

<u>**funct**</u> $ILS \;\equiv$
  $P := initialise;$

```
3      while ¬contains_solution(P) do
4          P := move(P);
5          evaluate(P);
6          P := select(P);
7      od
8  end
```

In algorithm 5.1 we see that the while-loop from line 3 to 7 iteratively applies the $move$-operator to a population of candidate solutions (P). The population is randomly initialised in line 2. The algorithm is terminated when a solution is found. Because some problem instances are unsolvable, a maximum number of iterations is commonly used to stop the algorithm as well. The $move$-operator of the ILS algorithm (line 4) modifies these candidate solutions using a heuristic embedded in the operator. The $select$-operator (line 6) then selects candidate solutions for the next iteration. Selection of the population for the next iteration of the algorithm is based on the evaluation of the population, implemented in the $evaluate$-operator, also called the objective function.

Many different implementations of the ILS algorithm have been proposed. Different selection methods provide different operators based on the notion of the selection pressure. Selection pressure is used to express the strength of the selection. High selection pressure is exerted when only the best candidate solutions are selected, no selection pressure is exerted when candidate solutions are selected uniform randomly. Selection is related to the problem by the objective function. The best candidate solutions are selected, for example, by ordering the population according to the value given by the objective function. The best candidate solution is then the first candidate solution in the ordering. Different problems have different objective functions and sometimes different objective functions exist for a single problem.

The move operator includes a heuristic, or rule-of-thumb, and is used to search through the search space of the problem. This heuristic can be deterministic or non-deterministic. The move operator usually focusses on part of the problem, a sub-problem, trying to solve it every time the heuristic is used. At each iteration of the algorithm different sub-problems can be solved. The choice of which sub-problem to solve can be made randomly but usually a heuristic is used for this as well. ILS is closely related to neighbourhood search. In neighbourhood search a sub-problem is chosen and all possible solutions for the sub-problem are generated. The select operator then selects the best solution, i.e., candidate solution, that was generated. When two best candidate solutions with equal quality have been generated, one of them is selected at random. For example, a move-operator for the CSP can be implemented by selecting a variable of the CSP instance and generating candidate solutions were this variable is labelled with all possible values in the domain of the variable. The selection operator then selects the candidate solution with the least number of constraint violations. The set of candidate solutions with a different label for a single variable can been seen as the neighbourhood of the original candidate solution. The name neighbourhood search stems from the fact that the move operator searches through the neighbourhoods in the chain of candidate solutions in order to find a solution.

An example of an ILS algorithm is the *Simulated Annealing* algorithm [1]. Simulated

Annealing was introduced as a generalisation of a Monte Carlo method for examining the equations of state and frozen states of n-body systems [63]. The concept is based on the manner in which liquids freeze or metals re-crystalise in the process of annealing. In an annealing process, a melt, initially at high temperature and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more and more ordered and approaches a "frozen" state at its lowest temperature. The process can be thought of as an adiabatic approach to the lowest energy state. If the initial system temperature is too low or cooling is done insufficiently slowly, the system may become quenched, forming defects or freezing out in meta-stable states, i.e., trapped in a local minimum energy state. Simulated Annealing is an example of an ILS algorithm with adaptive selection pressure regulated by temperature, applied on a population of candidate solutions altered by a move operator specific to a problem. For different problems, different move operators can be used.

In the next section two examples of general ILS algorithms are given: the *Random Search Algorithm* and the *Hill Climber with Restart Algorithm*. Both algorithms will be used as benchmark algorithms in the rest of the thesis. In the last section of this chapter, evolutionary algorithms will be introduced. A basic evolutionary algorithm, called the *Intuitive Evolutionary Algorithm* will be introduced as a benchmark for the other evolutionary algorithm introduced later in this thesis.

5.1 The *Random Search Algorithm* and the *Hill Climber with Restart Algorithm*

Two Iterated Local-Search algorithms will be introduced in this section: the *Random Search Algorithm* (*RSA*) and the *Hill Climber with Restart Algorithm* (*HCAWR*). The *Random Search Algorithm* is a very simple algorithm and throughout the rest of the thesis it will be used to distinguish the CSP instances that are easy to solve from the ones that are hard to solve. The *Hill Climber with Restart Algorithm* is more powerful and it will be used as a performance benchmark for the evolutionary algorithms in the thesis.

5.1.1 The *Random Search Algorithm*

The *Random Search Algorithm* is to the ILS algorithms what a brute-force algorithm is to the classical algorithms. It tries to solve a problem by repeatedly checking if randomly instantiated candidate solutions are solutions to the problem. A randomly instantiated candidate solutions for the CSP is a candidate solution were all variables are labelled with a uniform randomly chosen value from the variable's domain.

The *Random Search Algorithm* does not include an imbedded heuristic to guide the search, nor does it have memory or a selection operator. It is also possible to randomly instantiate a candidate solution that has been checked before. At the beginning of the search, the probability of 'rechecking' a candidate solution is small, but as the search continues, and more and more (unique) candidate solutions have been checked, this probability increases. The *Random Search Algorithm* is not a complete algorithm and will search for a solution indefinitely when the problem is unsolvable. A maximum number of candidate solutions that the *Random Search Algorithm* is allowed to check is therefore also used to terminate the search.

Like the brute-force algorithm for classical algorithms, the *Random Search Algorithm* has a low probability of finding a solution in reasonable time if the complexity of the problem is non-trivial. The usefulness of the *Random Search Algorithm* is therefore limited. In this thesis, the *Random Search Algorithm* is used to determine which constraint satisfaction problems are trivial or not. It is also used to provide a minimum performance for the other algorithms.

Algorithm 5.2 shows the pseudo-code of the *Random Search Algorithm*. It shows that the *Random Search Algorithm* has no selection operator. As the $initialise$ method produces randomly instantiated candidate solutions, it replaces the $move$ operator in line 5. Added are the $max\_evaluations$ parameter and the $evaluations$ variable in order to terminate the algorithm after a maximum number of candidate solutions have been checked. The check is made by the while statement (line 4). The $evaluate$ operator has been changed to return the number of evaluations necessary to evaluate the population. This is usually equal to the size of the population. If the population consists of only a single candidate solution, the maximum number of evaluations is equal to the number of iterations.

Algorithm 5.2: The *Random Search Algorithm*

1 **funct** *RSA*$(max\_evaluations) \equiv$
2   $evaluations := 0;$
3   $P := initialise;$
4   **while** $\neg contains\_solution(P) \vee evaluations < max\_evaluations$ **do**
5       $P := initialise;$
6       $evaluations := evaluations + evaluate(P);$
7   **od**
8 **end**

5.1.2 The *Hill Climber with Restart Algorithm*

The *Hill Climber with Restart Algorithm* is an example of a standard Iterated Local-Search algorithm. After initialising a population randomly, the *Hill Climber with Restart Algorithm* will solve a problem by repeatedly applying a heuristic move operator and selecting the best candidate solution for the next iteration. The *Hill Climber with Restart Algorithm* is not a complete algorithm and a maximum number of candidate solutions that it is allowed to check is therefore set as a parameter. The *Hill Climber with Restart Algorithm* terminates when either a solution of the problem is found or when the maximum number of candidate solutions is checked.

For the constraint satisfaction problem, the *Hill Climber with Restart Algorithm* initialises a candidate solution by labelling each variable of the candidate solution with a random value in the variable's domain. The most commonly used move operator selects a variable in the candidate solution uniform randomly and then generates the candidate solutions where that variable is labelled with all possible values in the domain of the variable. These candidate solutions are then added to the population. The selection operator then selects the candidate solution from the population which violates the least number of constraints of the CSP.

The *Hill Climber with Restart Algorithm* is an example of a neighbourhood search algorithm. A problem with using neighbourhood search is that it can become stuck in a local optimum. This happens when the neighbourhoods of all variables of the problem have been examined. Because all value combinations of these variables have to be checked, this takes a large number of search steps when the number of variables of the problem is large and/or the domains of these problems are large. Since the neighbourhood of a candidate solution depends on all values of the variables in the candidate solution, two candidate solutions in which only a single variable is labelled differently therefore have different neighbourhoods. When the neighbourhoods of all value-combinations of the variables have been examined, the *Hill Climber with Restart Algorithm* will revert to re-examining candidate solutions that have been checked already. When this happens, the population maintained by the *Hill Climber with Restart Algorithm* is said to have converged on a local optimum and the algorithm is said to be stuck in a local optimum. At this point, the *Hill Climber with Restart Algorithm* will be unable to proceed to a global optimum on its own.

In order for the *Hill Climber with Restart Algorithm* to escape a local optimum, a restart strategy is used: during the search, the *Hill Climber with Restart Algorithm* is restarted with a new, randomly generated, population, and the search for the global optimum is renewed. Different restart strategies can be applied, depending mostly on when to restart the algorithm. We have implemented a naive restart strategy, were the *Hill Climber with Restart Algorithm* is restarted after a preset number of iterations.

Algorithm 5.3 shows the pseudo-code of the *Hill Climber with Restart Algorithm* with this restart strategy. Like the *Random Search Algorithm*, the *Hill Climber with Restart Algorithm* also has a parameter called $max\_evaluations$ determining the maximum number of candidate checks allowed. The variable is checked against the $evaluations$ parameter in the while statement (4). Again the $evaluate$ operator returns the number of evaluations necessary to evaluate the population, usually equal to the size of the population. The $move\_hill\_climber$ described earlier replaces the $move$ operator in line 9. The restart strategy is implemented by adding the $restart\_interval$ parameter and the if-then-else statement. After an interval of $restart\_interval$ evaluations have been performed, the population is replaced by a new, randomly initialised, population (line 7). No more modification is then done, as it is possible to find a solution in the new population. The mod-operator returns the remainder of the division of $iteriations$ and $restart\_interval$. If $iterations$ is a natural multiple of $restart\_interval$, the mod-operator returns zero. It is possible that for certain combinations of population size and $restart\_interval$ values, the mod is not exactly zero while a restart of the algorithm is still necessary. When the number of evaluations for each iteration is equal to the population size, line 5 should then be replaced with **if** $evaluations > 0 \wedge evaluations \bmod restart\_interval < |P|$.

Algorithm 5.3: The *Hill Climber with Restart Algorithm*

```
1  funct HCAWR(max_evaluations, restart_interval) ≡
2    evaluations := 0;
3    P := initialise;
4    while ¬contains_solution(P) ∨ evaluations < max_evaluations do
5          if evaluations > 0 ∧ evaluations mod restart_interval = 0
6             then
7                   P := initialise;
8             else
9                   move_hill_climber(P);
10         fi
11         evaluations := evaluations + evaluate(P);
12         P := select(P);
13   od
14 end
```

5.2 Evolutionary Algorithms

Evolutionary algorithms are based on the evolution paradigm. First described by C. Darwin in “The Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life.” ([21]), the most widely accepted collection of evolutionary theories today is the neo-Darwinian paradigm. Neo-Darwinian theory arguments that the history of life can be fully accounted for by physical processes operating on and within populations and species ([47]).

The processes described in the neo-Darwinian paradigm are reproduction, mutation, competition, and selection. *Reproduction* is an obvious property of extant species. It is accomplished through the transfer of an individual’s genetic material to progeny. *Mutation* is guaranteed, in that replication errors during information transfer will necessarily occur. *Competition* is the consequence of expanding populations in a finite resource space. *Selection* is the inevitable result of competitive replication as species fill the available space. Evolution becomes the inescapable result of interacting basic physical statistical processes ([49, 88, 4] and others).

In [62], E. Mayr summarised some of the more salient characteristics of the neo-Darwinian paradigm:

1. The individual is the primary target of selection.
2. Genetic variation is largely a chance phenomenon, stochastic processes play a significant role in evolution.
3. Genotypic variation is largely a product of recombination and “only ultimately of mutation”.

4. “Gradual” evolution may incorporate phenotypic discontinuities.

5. Not all phenotypic changes are necessarily consequences of *ad hoc* natural selection.

6. Evolution is a change in adaptation and diversity, not merely a change in gene frequencies.

7. Selection is probabilistic, not deterministic.

Simulations of evolution rely on these foundations [38, 32, 8]. They are translated into algorithms using the common underlying idea of all evolutionary algorithms: given a population of individuals, the environmental pressure causes natural selection (survival of the fittest) which causes a rise in the overall fitness of the population.

That such a process can be used for optimisation is easy to see. Given an objective function, a set of candidate solutions can be randomly created. By applying the objective function, an abstract fitness measure can be calculated for all candidate solutions in the set. Based on this fitness, some of the better candidate solutions are chosen to seed the next generation by applying recombination and/or mutation.

Recombination is then an operator applied to a number of candidate solutions (usually two), called parents, which results in a number of candidate solutions, called children. Mutation is usually a unary operation applied to one candidate solution which produces as a result a single new candidate solution. The candidate solutions produced by recombination and mutation form an offspring population which competes, based on their fitness, with the parent population for a place in the next generation. This process is iterated until either a solution is found or a previously set computational limit is reached, usually, a maximum number of candidate solutions that are examined.

In this process, selection acts as a force pushing quality, while the variation operators, recombination and mutation, create the necessary diversity. Their combined application leads to improving fitness values in consecutive populations, approximating optimal fitness values closer and closer.

Many components of the evolutionary process are stochastic. In selection, fitter individuals have a higher chance to be selected than less fit ones, but typically, even weak individuals have a chance to become a parent or to survive. Recombination is stochastic as, in general, the choice of which variables of the candidate solution will be recombined is made randomly. Similarly for the mutation operator, the variables that are to be mutated, and the values that they are taking are chosen randomly.

Evolutionary algorithms are studied by the Evolutionary Computation research field. Over the years, four main dialects within the evolutionary computation field have been established: *Evolutionary Strategies*, *Evolutionary Programming*, *Genetic Algorithms*, and *Genetic Programming*. The differences between the four dialects are characterised by the typical representations, the methods for producing random variance in the population, and the method employed for selecting parents. A discussion on these differences can be found in [32]. Here, it suffices to say that the algorithms discussed in this thesis are most closely related to *Genetic Algorithms*.

5.2.1 The *Intuitive Evolutionary Algorithm*

The *Intuitive Evolutionary Algorithm* is used as a benchmark evolutionary algorithm for the other evolutionary algorithms in this thesis. It is specifically designed to solve constraint satisfaction problems and is: easy to understand, has decent performance and has no major alterations to the canonical evolutionary algorithm described above.

The pseudo-code of the *Intuitive Evolutionary Algorithm* is given in algorithm 5.4. From the similarities between algorithm 5.1 and 5.4 it is easy to see that evolutionary algorithms are part of the Iterated Local-Search group. Two differences are apparent: The $select$ operator from algorithms 5.2 and 5.3 is split into two selection operators, $select\_parents$ and $select\_survivors$, and the $move$ operator is split into a $crossover$ and a $mutate$ operator.

Algorithm 5.4: The *Intuitive Evolutionary Algorithm*

```
1  funct IEA(max_evaluations) ≡
2    evaluations := 0;
3    P := initialise;
4    while ¬contains_solution(P) ∨ evaluations < max_evaluations do
5          S := select_parents(P);
6          S := crossover(S);
7          S := mutate(S);
8          evaluations := evaluations + evaluate(S);
9          P := select_survivors(P, S);
10   od
```

The split in the $select$ operator is necessary because evolutionary algorithms apply the $crossover$ and $mutate$ operators on just a part of the population called the parent population. The candidate solutions in the parent population are selected with replacement. The $crossover$ operator typically takes two candidate solutions from the parent population and produces two candidate solutions from them. Many different $crossover$ operators have been proposed. The candidate solutions produced by the $crossover$ operator are called the children of the operator, the population of all children is called the child population. It is used as a parent population for the $mutate$ operator. The $mutate$ operator takes a single parent candidate solution and produces a single child candidate solution. The $initialise$ operator initialises the population randomly, just as in algorithms 5.2 and 5.3, the $evaluate$ operator is also the same as in those two algorithms. The conditional statement in the while loop (line 4) is called the stop-condition of the algorithm.

In evolutionary algorithms it is customary to use the term *chromosome* for *candidate solution* and *gene* and *allele* for *variable* and *value* respectively. The term *individual* is commonly used as a synonym for *chromosome* but we will use in its more precise meaning, which is to refer to a pair consisting of a candidate solution and its fitness value. One iteration of an evolutionary algorithm is often called a *generation*. The $crossover$- and the $mutation$-operators together are called the genetic- or variation-operators of an evolutionary algorithm.

Innards of the *Intuitive Evolutionary Algorithm*

This section will describe how the *Intuitive Evolutionary Algorithm* is implemented to solve constraint satisfaction problems. In [48], Holland suggested that, for genetic algorithms, candidate solutions should be implemented using a binary representation. For the CSP this would entail the encoding of each value as a binary vector. The complete candidate solution would then be the concatenation of these vectors in order. This representation has been criticised as being cumbersome and impractical for problems including real values. For the CSP especially, it was found that representing the candidate solutions as a vector of values, without encoding, is more practical with no adverse affects on the performance of the algorithm. As such, the *Intuitive Evolutionary Algorithm* uses this representation for its individuals. This representation is denoted as an *ordered set of values*. The individuals are initialised by uniform randomly selecting a value from the domain of each variable in the CSP. A population is then a set of these individuals.

The fitness value of an individual is calculated by the objective function. In algorithm 5.4 this is done using the $evaluate$ operator. This operator evaluates all individuals in the population. The fitness value of an individual is commonly referred to as the *fitness* of an individual. The fitness of an individual is used by the selection operators for selecting certain individuals over others for the next generation. The selection operators thus determines the direction of the search of an evolutionary algorithm. An objective function for an evolutionary algorithm solving a CSP has to be able to determine if a candidate solution is a solution to the CSP, since at this point the search can stop. However, since the CSP is a satisfaction problem, for an evolutionary algorithm, only determining whether or not a candidate solutions is a solution is not enough. An objective function also has to be able to distinguish which of two candidate solutions is better without them being solutions to the CSP. Two commonly used methods for this have been proposed ([17]):

1. Assign a fitness value based on the number of constraints that the individual violates; and

2. Assign a fitness value based on the number of variables that violate a relevant constraint

An individual is then a solution when either no constraints are violated or when no variables violate their relevant constraints. Both objective functions are to be minimised. Given a CSP $\langle X, D, C \rangle$, $s = (\langle x_1, v_1 \rangle, \ldots, \langle x_{|X|}, v_{|X|} \rangle)$ a candidate solution, c_i a constraint in C, and C^j the set of constraints relevant to x_j, the two objective functions f_1 and f_2 are defined as follows:

$$f_1(s) = \sum_{i=1}^{|C|} \chi(s, c_i) \tag{5.1}$$

where

$$\chi(s, c_i) = \begin{cases} 1 & \text{if } violates(s, c_i) \\ 0 & \text{otherwise.} \end{cases} \tag{5.2}$$

and

$$f_2(s) = \sum_{j=1}^{|X|} \chi(s, C^j) \tag{5.3}$$

where

$$\chi(s, C^j) = \begin{cases} 1 & \text{if } \exists c \in C^j : violates(s, c) \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

Objective function f_1 provides more information than f_2. This is obvious when the range of the fitness values of the two objective functions are compared. The range of the fitness values of f_1 is $\langle 0, |C| \rangle$, the range of the fitness values of f_2 is $\langle 0, |X| \rangle$. The number of constraints in a CSP is calculated using $p_1 \cdot \frac{1}{2}|X| \cdot (|X| - 1)$, therefore when $p_1 \cdot \frac{1}{2} \cdot |X| \cdot (|X| - 1) > |X|$, f_1 will provide more information. The ranges of the fitness values of both objective functions are equal when $p_1 \cdot (\frac{1}{2}|X| - \frac{1}{2}) = 1$. For example, for a CSP with 10 variables, the f_1 objective function will provide more information when the density is between 0.05 and 0.95. Because the fitness values are calculated over the constraints of the CSP, however, the f_1 objective function will use more conflict checks per evaluation. The *Intuitive Evolutionary Algorithm* will use the first objective function (f_1).

Many different parent selection operators have been proposed for evolutionary algorithms. In various ways, all try to maintain a balance between the selection of good individuals to for further development and lesser individual in order to maintain diversity of the population. The *Intuitive Evolutionary Algorithm* uses a parent selection operator based on *linear ranking selection* ([87]). Linear ranking selection orders (ranks) the individuals in the population by their fitness values. Individuals are then uniform randomly selected based on their rank in the ordering by generating a pseudo-random number between 0 and $pop\_size - 1$, where pop_size stands for the size of the population. Since most pseudo-random number generators only generate numbers in the range $[0, 1\rangle$, the rank is calculated by multiplying the random number by pop_size and rounding it down to the nearest integer number:

$$i = \lfloor pop\_size \cdot random \rfloor \tag{5.5}$$

where i is the rank in the ordered population and $random$ a pseudo-random number in the range $[0, 1\rangle$. $\lfloor \cdot \rfloor$ denotes that the number is rounded down to the nearest natural

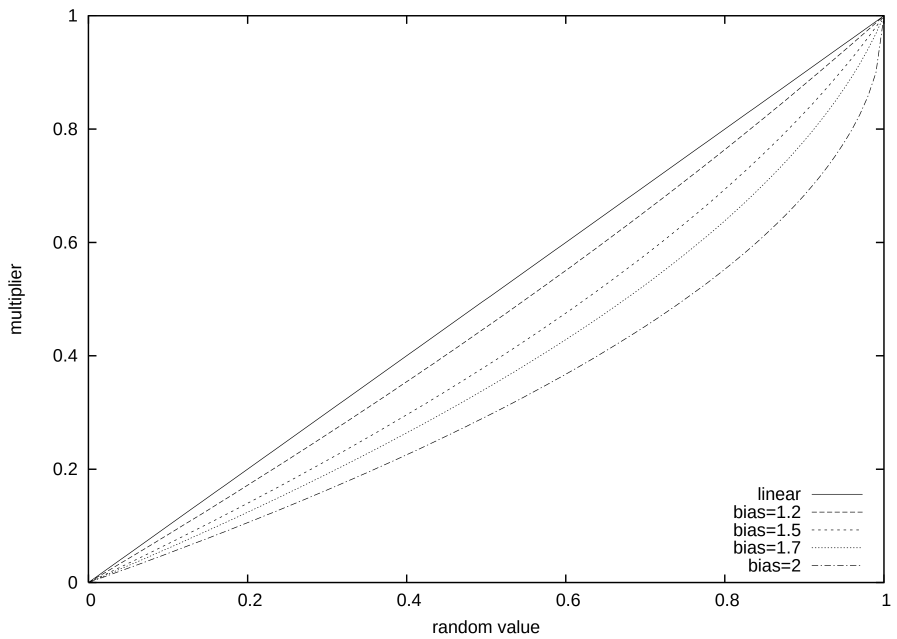

Figure 5.1: Biased ranking multiplier plotted against *random*-values for *bias* $\in \{1.0\text{(linear)}, 1.2, 1.5, 1.7, 2\}$.

number. Selection pressure in linear ranking selection is exerted through the random selection of *ranked* individuals.

The *Intuitive Evolutionary Algorithm* changes the linear ranking selection operator by adding a bias so that better individuals are more often selected. The operator is called the *biased ranking* selection operator. The amount of bias is set by a bias-parameter for the operator: $bias$. The range of $bias$ is between 1 (no bias, or linear ranking selection) and 2 (strong bias), inclusive. Which individual is selected is calculated by the following equation:

$$i = \left\lfloor pop\_size \cdot \frac{bias - \sqrt{bias^2 - (4 \cdot (bias - 1)) \cdot random}}{2 \cdot (bias - 1)} \right\rfloor \tag{5.6}$$

where i is the rank in the ordered population, and $random$ and $\lfloor \cdot \rfloor$ the same as in equation 5.5.

The effect of different values for the bias-parameter is show in Figure 5.1. It shows the ranking multiplier $(bias - \sqrt{bias^2 - (4 \cdot (bias - 1)) \cdot random})/(2 \cdot (bias - 1))$ (y-axis) applied to the population size (equation 5.6) for different values of $bias$ for the range of possible $random$ values (x-axis). The line “linear”, when $bias = 1$, shows that no bias is applied and every individual has the same chance of being selected. When $bias$ is increased, the range of $random$ where higher ranked individuals

IEA	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	f_1
Crossover operator	Uniform Random Crossover
Mutation operator	Uniform Random Mutation
Parent Selection	Biased Ranking
Survivor Selection	Elitist Replace Worst
Other Functions	None

Table 5.1: Characteristics of the *Intuitive Evolutionary Algorithm*.

are chosen increases while the range of $random$ where lower ranked individuals are chosen decreases.

The survivor selection operator merges the child-population of the genetic operators (S) with the population of the evolutionary algorithm (P). The *Intuitive Evolutionary Algorithm* uses an *elitist replace worst survivor selection* operator. A survivor selection operator is called elitist when is preserves individuals from the population with the best fitness value. In the *Intuitive Evolutionary Algorithm* only a single individual from the population is preserved. The other individuals from the population are replaced by individuals from the child population when their fitness values are worse. The survivor selection operator in the *Intuitive Evolutionary Algorithm* maintains the size of population. An evolutionary algorithm in which recombination of less than the whole population is performed every generation is said to employ the *steady state* evolutionary model.

The genetic operators used in the *Intuitive Evolutionary Algorithm* are called the *uniform random crossover* operator and the *uniform random mutation* operator. The uniform random crossover operator takes two parent individuals and randomly swaps each value between them, producing two child individuals. Uniform random mutation is also called k/l-mutation. It takes a single parent individual and changes each value with probability p, called the mutation rate. It takes its name from the two parameters to calculate the mutation rate: l for the number of values of the individuals, here the number of variables of the CSP to solve, and k the parameter to determine the mutation rate using the equation: $p = \frac{k}{l}$. Much theoretical and empirical research has been done on the best mutation rate setting (see for example [36, 43, 79, 67]) for different evolutionary algorithms for different problems. Through experimentation we found that $k = 1$ is a near optimal value for the mutation rate for the *Intuitive Evolutionary Algorithm*, constraint satisfaction problem combination. The value in the individual is changed to another value in the domain of its variable.

The characteristics of all evolutionary algorithms proposed in the thesis will be summarised in *characteristics tables*. The characteristics table of the *Intuitive Evolutionary Algorithm* is shown in Table 5.1.

Chapter 6

Performance Measures and Experimentation

In this chapter the *Random Search Algorithm*, the *Hill Climber with Restart Algorithm*, and the *Intuitive Evolutionary Algorithm* algorithms will be used as an example of our method of experimentation. First we introduce the performance measures that will be used throughout the thesis and how they are displayed in tables and figures. The measurements of the three algorithms will be shown next. In the third and final section of the chapter we show how the results are compared and how conclusions can be drawn from them with a certain degree of accuracy.

6.1 Performance Measures

The classical algorithm described earlier only needed a single performance measure, the number of conflict checks needed to find a solution if the problem instance is solvable or the number of conflict checks needed to determine if a problem instance is unsolvable. Because non-deterministic algorithms are not complete, the conflict checks performance measure does not give enough information. If, for example, a non-deterministic algorithm does not find a solution during a run, this does not imply that the problem instance the algorithm was trying to solve is unsolvable. This can only be estimated with some degree of certainty with a very long run or a large number of shorter ones and even then, there is the possibility of not finding a solution when there is one. Since in this thesis we use a test-set that contains only solvable CSP instance, this experiment is actually unnecessary, however, this does not mean that multiple runs on a single instance are also unnecessary because multiple runs will provide an estimate of the overall performance of the algorithm. An accurate estimate of the overall performance of the algorithm can be given by running the algorithm multiple times on the same (set of) problem instances and then averaging the performance measures over the number of runs. The accuracy of the estimate increases when the number of runs increases.

This section will define a number of performance measures. The measures will be used to assess the performance of the algorithms on three properties:

1. The effectiveness; which determines how good an algorithm is in finding a solution;
2. The efficiency; which determines how fast an algorithm can find a solution; and
3. The behaviour: which gives an insight in how an algorithm finds a solution.

Behavioural measures can also give an explanation on why one algorithm outperforms another.

6.1.1 Success Rate

The Success Rate (*SR*) of an algorithm is calculated by dividing the number of successful runs of an algorithm by the total number of runs. A successful run of an algorithm is a run where the algorithm found a solution to the problem. The range of the *SR* measure is between 0 and 1, but is sometimes expressed as a percentile. If the *SR* is 0, no solutions were found, if it is 1, all runs were successful. The *SR* is a measure of the effectiveness of the algorithm.

The *SR* measure is the most important measure when we compare two algorithms. An algorithm with a higher *SR* finds more solutions than an algorithm with a lower *SR*, and finding solutions is, after all, what the algorithm is designed to do. The accuracy of the *SR* measure is influenced by the total number of runs, more runs provide a more accurate approximation of the *SR* of the algorithm. When the difference between the *SR* of two algorithms is small, it does not necessarily mean that the algorithm with the best *SR* outperforms the other algorithm. The difference can also be caused by the inaccuracy of the measure, properties of the test-set used, and random influences. Further analysis is then necessary.

6.1.2 Average Number of Evaluations to Solution

The average number of evaluations to solutions (*AES*) of an algorithm is calculated by the average number of evaluations over all successful runs. The number of evaluations is calculated by counting the number of times that the evaluate operator was used by the algorithm. If a run is unsuccessful, *AES* is undefined. The *AES* is a measure of the efficiency of the algorithm.

The *AES* measure is used as a secondary measure for comparing two algorithms. When two algorithms have approximately the same *SR*, the *AES* measure is used to determine which algorithm is more efficient. The algorithm with the lower *AES* is more efficient than the algorithm with a higher *AES*.

6.1.3 Conflict Checks

The number of conflict checks needed to find a solution (*CC*) measure is calculated by the average number of conflict checks over all successful runs. The number of conflict checks is calculated by counting the number of times that a compound label is tested to be in a constraint of the CSP. If a run is unsuccessful, *CC* is undefined. The *CC* measure is a measure of the efficiency of the algorithm.

The *CC* measure is used as a more fine grained efficiency measure or to compare the performance of a non-deterministic algorithm with a classical algorithm. The *CC* measure is more precise than the *AES* because it counts the conflict checks used while the *AES* counts the evaluations. Because different evaluation operators use different amounts of conflict checks and evaluations of different candidate solutions also use different amounts of conflict checks, the difference between two algorithms can be quite large.

The *CC* measure also accounts for the "hidden work" done by the algorithm. Hidden work is defined as the number of conflict checks performed by the algorithm outside the evaluation operator. The efficiency of the evaluation operator can be approximated by dividing the *CC* by the *AES*. This can only be an indication of the efficiency because it leaves out the hidden work performed outside the evaluation operator.

6.1.4 Unique Individuals Checked

The number of unique individuals checked (*UIC*) measure is calculated by counting the number of unique candidate solutions that were evaluated during the run. The *UIC* measure is a behavioural measure and is measured at intervals during a run. When the *UIC* measure is applied to a number of runs, the measure is averaged over all runs at each interval. The interval over which the measure is calculated is usually every 100 or 1000 evaluations, depending on the maximum number of evaluations allowed.

For a single run, the *UIC* consists of a monotonic increasing sequence of values. When the algorithm has not converged on a local optimum it consists of a strict monotonic increasing sequence of values. When the *UIC* is averaged over all runs, it does not have to consist of a monotic sequence of values, as smaller numbers of unique individuals can occur when one of the runs is successful and the remaining runs have an average *UIC* that is smaller than the average *UIC* including the successful run. The *UIC* measure is depicted as a plot where on the x-axis the number of evaluations and on the y-axis the (average) *UIC* is shown. The line where every evaluated candidate solution is unique is added as a reference.

6.1.5 Mean Best Fitness and Mean Champion Error

The mean best fitness (*MBF*) measure is calculated by averaging the fitness value of the best candidate solution in the population over a number of runs at a given moment. Moments are specified via our notion of time, measured by performed fitness evaluations. The *MBF* measure is depicted as a plot where on the x-axis the number of evaluations and on the y-axis the *MBF* measure is shown. The *MBF* measure depends on the fitness function. This makes comparing two algorithms with different fitness functions difficult which is why in the same plot the champion error is added.

The mean champion error (*MCE*) measure is calculated by averaging the number of violated constraints of the best candidate solution (the champion) in the population, again over a number of runs at a given moment. Just as the *MBF* measure, the intervals are determined using the number of performed fitness evaluations. This measure is independent of the evaluation operator used. A plot where both the *MBF* and the *MCE* measure are shown uses the left-hand y-axis for the *MBF* measure and the right-hand y-axis for the *MCE* measure.

The interval over which both measures are commonly used is 100 or 1000 evaluations. Both the *MBF* and the *MCE* measures are behavioural measures.

6.2 Experimentation

All experiment in this thesis will be performed on the test-set generated in Chapter 4. 10 independent runs on all 1475 instances in the test-set will be performed. Although this might seem like a low number of runs, performing 10 independent runs on 25 instances for each density-tightness combination in the test-set provides 250 sample points for each density-tightness combination. As there are 59 density-tightness combinations in the test-set this amounts to a total of 14750 runs performed for each algorithm. The *SR* is calculated over all 250 sample points for each density-tightness combination, the *AES* and *CC* measures are calculated over successful runs only. The *UIC*, *MBF*, and *MCE* measures are calculated at an interval of 1000 evaluations during each run. All algorithms use a population size of 10 candidate solutions for all runs. A maximum number of 100000 evaluations is allowed for each algorithm. With a population size of 10 candidate solutions this allows for approximately 10000 generations depending on the algorithm used.

The results of the experiments will be summarised by three tables and two plots of each algorithm. The tables show the *SR*, *AES*, and *CC* measures. Along the columns of the table the density is shown, along the rows the average tightness is shown. Density-tightness combinations not in the test-set are represented with a '-'. The density-tightness combinations in the mushy region are represented in the lowest row for each column in the tables. When the *AES* and *CC* measures exceed 100000000 evaluations and conflict checks respectively, they will be rounded to the nearest million with $\cdot 10^6$added. The two plots show the *UIC*, and the *MBF MCE* plots as explained earlier.

6.2.1 Results of the *Random Search Algorithm*

In Table 6.1 the parameters used for the experiments with *Random Search Algorithm* are shown. Table 6.2 shows the *SR* of the *Random Search Algorithm*. It shows that

RSA	
Population Size	10
Selection Siz	10
Maximum Number of Evaluations	100000

Table 6.1: Parameters of the *RSA*.

the *Random Search Algorithm* is unable to solve any CSP instance in the mushy region except for density-tightness combination $(0.1, 0.9)$ where 53.2% of the runs were successful. As the *Random Search Algorithm* searches for a solution by checking randomly instantiated candidate solutions, this rather poor performance was to be expected. Table 6.2 also shows that for a large portion of the solvable region in the test-set, *RSA* found a solution for all runs (a *SR* of 1.0). The instances in this region are obviously very easy to solve and should not be used to compare the performance of two algorithms. Table 6.2 also shows that the *SR* of the *Random Search Algorithm* drops off sharply after these easy instances. For the harder instances a more powerful search method is required.

Table 6.3 shows the *AES* of the *Random Search Algorithm*. For the density-tightness combinations where no runs were successful, the *AES* measure is undefined, indicated by *undef.*For the density-tightness combinations where all runs were successful the *AES* is low. The *AES* measure is inaccurate when the number of successful runs (the *SR*) is low. The *AES* increases when the complexity increases, indicating that more search was necessary. The only two exceptions are density-tightness combinations $(0.4, 0.6)$ and $(0.5, 0.5)$ but this is due to the low *SR* of these density-tightness combinations and the inaccuracy of the *AES*.

Table 6.4 shows the *CC* of the *Random Search Algorithm*. Just as with the *AES* measure, for density-tightness combinations where no runs were successful, the *AES* measure is undefined, indicated by *undef.*. The *CC* measure is also inaccurate when the *SR* for a density-tightness is low. Again, the *CC* increases when the complexity of the instances increases.

Figure 6.1 shows the *UIC* of the *Random Search Algorithm* for the density-tightness combinations in the mushy region. Throughout the thesis, whenever we display plots of results in the mushy region, we do so by displaying a group of nine plots. Each plot in the group displays the results of an experiment on the set of CSP instances of one of the density-tightness combinations in the mushy region. The plots are displayed in the following order: The top row, from left to right; density-tightness combinations $(0.1, 0.9)$, $(0.2, 0.9)$, and $(0.3, 0.8)$. The middle row, from left to right; density-tightness combinations $(0.4, 0.7)$, $(0.5, 0.6)$, and $(0.6, 0.6)$. The bottom row, from left to right; $(0.7, 0.5)$, $(0.8, 0.5)$, and $(0.9, 0.4)$.

The plots show that the *Random Search Algorithm* examines a unique individual almost every time a new individual is initialised. The chance of initialising a new individual that was already examined before is small but increases as more individuals are examined. After the maximum number of individuals allowed were examined, the chance of generating an individual that was already examined is approximately $\frac{10^4}{10^{10}} = \frac{1}{10000000}$. The *Random Search Algorithm* searches through almost the maximum search space allowed, unfortunately, most of the search space searched through is infeasible.

Figure 6.2 shows the *MBF* and *MCE* of the *Random Search Algorithm* for the density-tightness combinations in the mushy region. The straight lines through almost all plots indicate that no real search was performed. The exception is the plot for density-tightness combination $(0.1, 0.9)$ which shows a "saw-tooth" line for *MBF*. This is caused by the successful runs. When a runs are successful, the best fitness of the individuals in their populations is 0. When a runs is successful at the interval when the measure is taken this reduces the average mean best fitness value indicated by the spike downwards. When the next interval is calculated, the successful run is not included and the average mean best fitness is back at its former value. The spikes increase in depth because the average is taken over fewer values as more and more runs are successful and are left out. The spike is double the depth when two runs are successful at the same interval in the run.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	0.58	0.216	0.044
0.4	1.0	1.0	1.0	1.0	0.52	0.088	0.0	0.0	0.0
0.5	1.0	1.0	0.996	0.28	0.012	0.0	0.0	0.0	—
0.6	1.0	1.0	0.208	0.004	0.0	0.0	—	—	—
0.7	1.0	0.764	0.0	0.0	—	—	—	—	—
0.8	1.0	0.104	0.0	—	—	—	—	—	—
0.9	0.532	0.0	—	—	—	—	—	—	—

Table 6.2: *SR* of the *Random Search Algorithm*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	12	14	18	29	42	58	95
0.2	10	15	30	82	199	459	1435	3268	11966
0.3	13	37	185	644	3724	14789	40496	42819	46935
0.4	20	116	1440	9780	46054	44260	*undef.*	*undef.*	*undef.*
0.5	51	536	17410	45909	26007	*undef.*	*undef.*	*undef.*	—
0.6	124	3724	50477	44650	*undef.*	*undef.*	—	—	—
0.7	465	38981	*undef.*	*undef.*	—	—	—	—	—
0.8	4615	47010	*undef.*	—	—	—	—	—	—
0.9	41146	*undef.*	—	—	—	—	—	—	—

Table 6.3: *AES* of the *Random Search Algorithm*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	40	62	89	120	171	278	406	566	937
0.2	35	65	146	394	978	2277	7152	16217	59915
0.3	37	118	601	2134	12241	49269	132841	142246	159318
0.4	47	288	3605	24587	116234	111178	*undef.*	*undef.*	*undef.*
0.5	97	1075	34225	92617	53995	*undef.*	*undef.*	*undef.*	—
0.6	205	6178	83460	76249	*undef.*	*undef.*	—	—	—
0.7	661	55714	*undef.*	*undef.*	—	—	—	—	—
0.8	5825	58842	*undef.*	—	—	—	—	—	—
0.9	46241	*undef.*	—	—	—	—	—	—	—

Table 6.4: *CC* of the *Random Search Algorithm*.

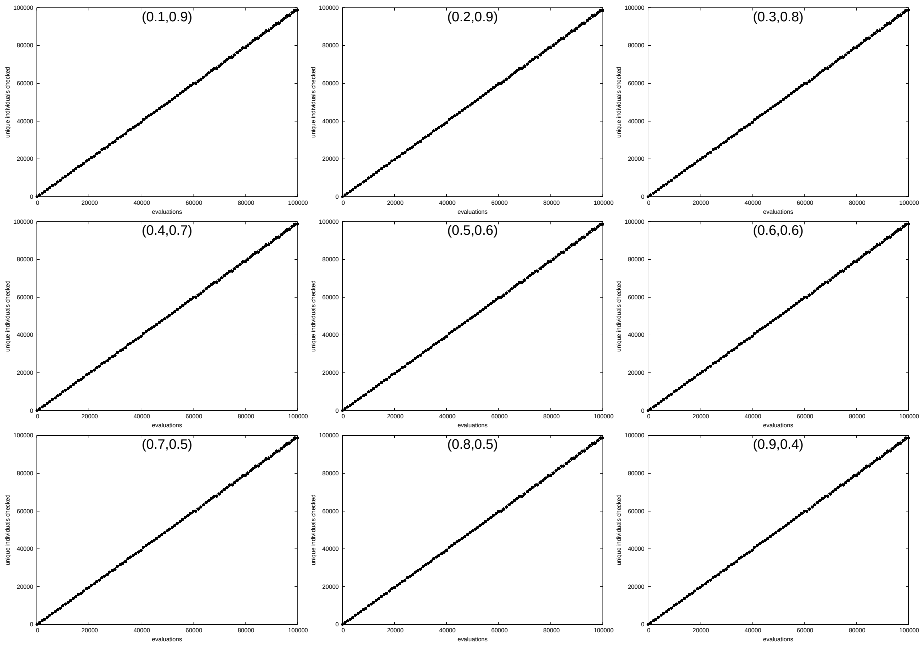

Figure 6.1: *UIC* of the *Random Search Algorithm*.

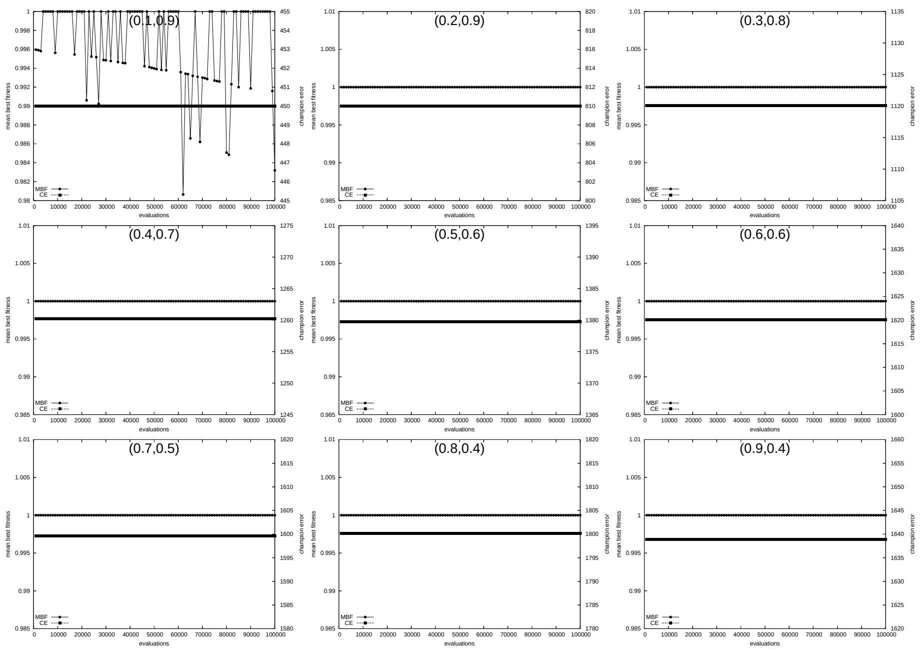

Figure 6.2: *MBF* and *MCE* of the *Random Search Algorithm*.

HCAWR	
Population Size	10
Selection Siz	10
Maximum Number of Evaluations	100000
Restart Interval	5000

Table 6.5: Parameters of the *HCAWR*.

6.2.2 Results of the *Hill Climber with Restart Algorithm*

In Table 6.5 the parameters used for the experiments with the *HCAWR* are shown. In order to find the restart interval of the *Hill Climber with Restart Algorithm*, a number of test experiments were done. It was found that after about 5000 evaluations the *Hill Climber with Restart Algorithm* converged to a local optimum and no new individuals would be examined. The restart interval was therefore set at 5000 evaluations. Table 6.6 shows the *SR* of the *Hill Climber with Restart Algorithm*. It shows that the *Hill Climber with Restart Algorithm* was successful in finding solutions in all runs.

Table 6.7 shows the *AES* of the *Hill Climber with Restart Algorithm*. Because all runs were successful, the *AES* measure for the *Hill Climber with Restart Algorithm* is reliable. This because the *AES* is an average measure and when all runs are successful its reliability doesn't suffer from a lack of samples. The table shows that the *Hill Climber with Restart Algorithm* needs relatively few evaluations to find a solution but that the *AES* increases as the complexity of the instances increases. This is substantiated by Table 6.8 which shows the *CC* of the *Hill Climber with Restart Algorithm*. Figure 6.3 shows the *UIC* plots of the *Hill Climber with Restart Algorithm* in the mushy region. The stepwise increase of the *UIC* is explained by the restart strategy. The steps have a length of 5000 evaluations. After this number of evaluations, the *UIC* does not increase, indicating a premature convergence to a local optimum. At this point the population is reinitialised randomly and the *UIC* increases again until 5000 evaluations later another convergence to a local optimum occurs, etc.

Figure 6.4 shows the *MBF* and *MCE* plots of the *Hill Climber with Restart Algorithm* in the mushy region. These plots too show stepwise changes because of the restart strategy used. The *MBF* of the population decreases stepwise while the *MCE* measure shows a spiked behaviour. The spikes occur when the reinitialised population includes not yet improved candidate solutions with a large error. The error is greatly decreased when after another interval the candidate solutions are improved by the move operator. The total number of evaluations of the *MBF* and *MCE* plots corresponds to the *UIC* plot, the spikes in the *MCE* line correspond to the steps in the *UIC* plot.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	—
0.6	1.0	1.0	1.0	1.0	1.0	1.0	—	—	—
0.7	1.0	1.0	1.0	1.0	—	—	—	—	—
0.8	1.0	1.0	1.0	—	—	—	—	—	—
0.9	1.0	1.0	—	—	—	—	—	—	—

Table 6.6: *SR* of the *Hill Climber with Restart Algorithm*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	13	17	22	25	33	33
0.2	10	14	23	31	39	48	54	60	69
0.3	12	24	40	50	62	73	129	235	579
0.4	17	34	55	70	281	720	2352	6203	15178
0.5	27	48	183	637	2747	7295	23718	17290	—
0.6	37	125	1112	3707	15487	18464	—	—	—
0.7	68	830	8744	16208	—	—	—	—	—
0.8	390	3487	15412	—	—	—	—	—	—
0.9	1858	9712	—	—	—	—	—	—	—

Table 6.7: *AES* of the *Hill Climber with Restart Algorithm*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	105	305	561	1207	2100	2743	4003	4288
0.2	98	628	1931	3206	4470	6101	7172	8448	10198
0.3	303	1949	4264	5943	7912	9736	18817	36017	93199
0.4	910	3286	6343	8660	39583	106039	360394	976505	$2 \cdot 10^6$
0.5	2229	5180	23882	87971	396324	$1 \cdot 10^6$	$4 \cdot 10^6$	$3 \cdot 10^6$	—
0.6	3541	15254	149729	516498	$2 \cdot 10^6$	$3 \cdot 10^6$	—	—	—
0.7	7554	107407	$1 \cdot 10^6$	$2 \cdot 10^6$	—	—	—	—	—
0.8	48309	454559	$2 \cdot 10^6$	—	—	—	—	—	—
0.9	234242	$1 \cdot 10^6$	—	—	—	—	—	—	—

Table 6.8: *CC* of the *Hill Climber with Restart Algorithm*.

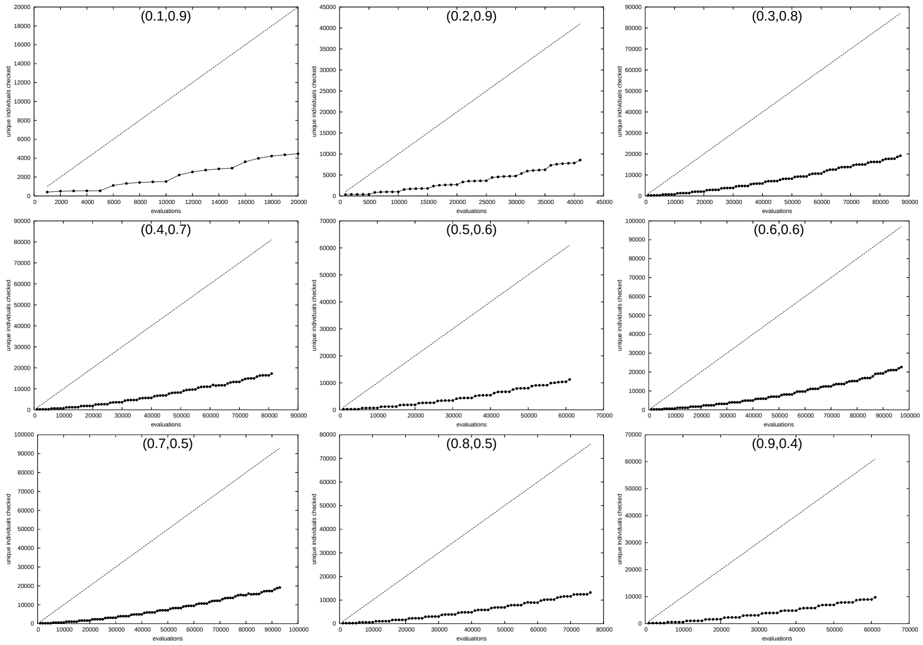

Figure 6.3: *UIC* of the *Hill Climber with Restart Algorithm*.

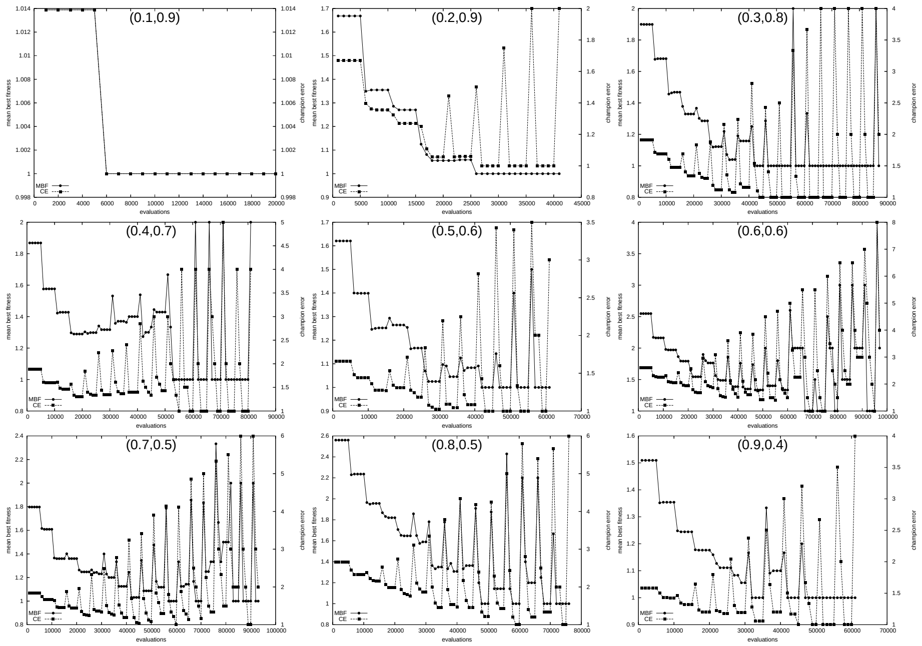

Figure 6.4: *MBF* and *MCE* of the *Hill Climber with Restart Algorithm*.

IEA	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	100000
Crossover Rate	1.0
Mutation Rate	0.1
Linear Ranking Bias	1.5

Table 6.9: Parameters of the *IEA*.

6.2.3 Results of the *Intuitive Evolutionary Algorithm*

In Table 6.9 the parameters used for the experiments with the *Intuitive Evolutionary Algorithm* are shown. Table 6.10 shows the *SR* of the *Intuitive Evolutionary Algorithm*. The *Intuitive Evolutionary Algorithm* finds solutions in the test-set throughout all density-tightness combinations although the performance is lower than the performance of the *Hill Climber with Restart Algorithm*. The *Intuitive Evolutionary Algorithm* has trouble finding solutions for the instances in the mushy region.

Table 6.11 shows the *AES* of the *Intuitive Evolutionary Algorithm*. The *AES* is higher than the *AES* of the *Hill Climber with Restart Algorithm*, especially when the hardness of the instances increases. Because the *SR* of the *Intuitive Evolutionary Algorithm* is low for these instances, the accuracy of the *AES* measure is also less than the accuracy of the *AES* measure for the *Hill Climber with Restart Algorithm*. This is substantiated by the *CC* of the *Intuitive Evolutionary Algorithm* shown in Table 6.12.

Figure 6.5 shows the *UIC* plots of the *Intuitive Evolutionary Algorithm* in the mushy region. The plots show that the *UIC* keeps increasing during the run but that the rate of increase decreases. Important to note is that during the run no premature convergence to a local optimum occurred.

Figure 6.6 shows the *MBF* and *MCE* plots of the *Intuitive Evolutionary Algorithm* in the mushy region. The *MBF* and *MCE* lines in the plots lie close together because the evaluation operator of the *Intuitive Evolutionary Algorithm* is actually an implementation of the *MCE* measure. The spikes in the plots for density-tightness combination $(0.1, 0.9)$ are caused by successful runs and the effect they have on the average taken for both methods. This effect is less for the other plots because the number of successful runs is less.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	0.992	0.972	0.824
0.4	1.0	1.0	1.0	1.0	0.98	0.872	0.576	0.292	0.088
0.5	1.0	1.0	0.996	0.916	0.596	0.252	0.06	0.088	—
0.6	1.0	1.0	0.876	0.476	0.108	0.068	—	—	—
0.7	1.0	0.892	0.328	0.108	—	—	—	—	—
0.8	0.98	0.584	0.064	—	—	—	—	—	—
0.9	0.808	0.156	—	—	—	—	—	—	—

Table 6.10: *SR* of the *Intuitive Evolutionary Algorithm*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	12	16	22	28	37	42
0.2	10	13	23	37	56	79	117	133	196
0.3	12	25	57	86	171	226	1319	2697	6510
0.4	18	46	135	337	2133	5282	10054	13766	20571
0.5	31	86	1300	3151	10545	10500	19471	12835	—
0.6	51	500	5138	15594	11971	9929	—	—	—
0.7	92	3499	10652	19965	—	—	—	—	—
0.8	2361	9272	16009	—	—	—	—	—	—
0.9	4775	18352	—	—	—	—	—	—	—

Table 6.11: *AES* of the *Intuitive Evolutionary Algorithm*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	91	153	221	372	584	892	1346	1719
0.2	51	117	327	662	1289	2128	3729	4785	8023
0.3	59	223	795	1557	3940	6111	42204	97081	266928
0.4	90	415	1893	6068	49053	142605	321729	495587	843407
0.5	155	778	18198	56724	242536	283500	623083	462060	—
0.6	254	4503	71936	280690	275336	268078	—	—	—
0.7	461	31492	149134	359367	—	—	—	—	—
0.8	11807	83447	224122	—	—	—	—	—	—
0.9	23873	165166	—	—	—	—	—	—	—

Table 6.12: *CC* of the *Intuitive Evolutionary Algorithm*.

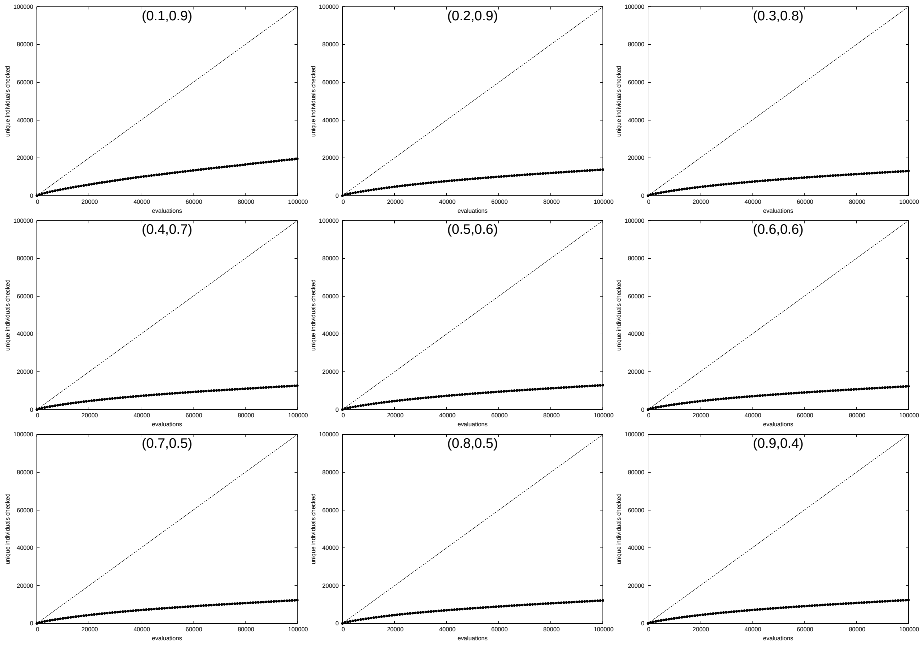

Figure 6.5: *UIC* of the *Intuitive Evolutionary Algorithm*.

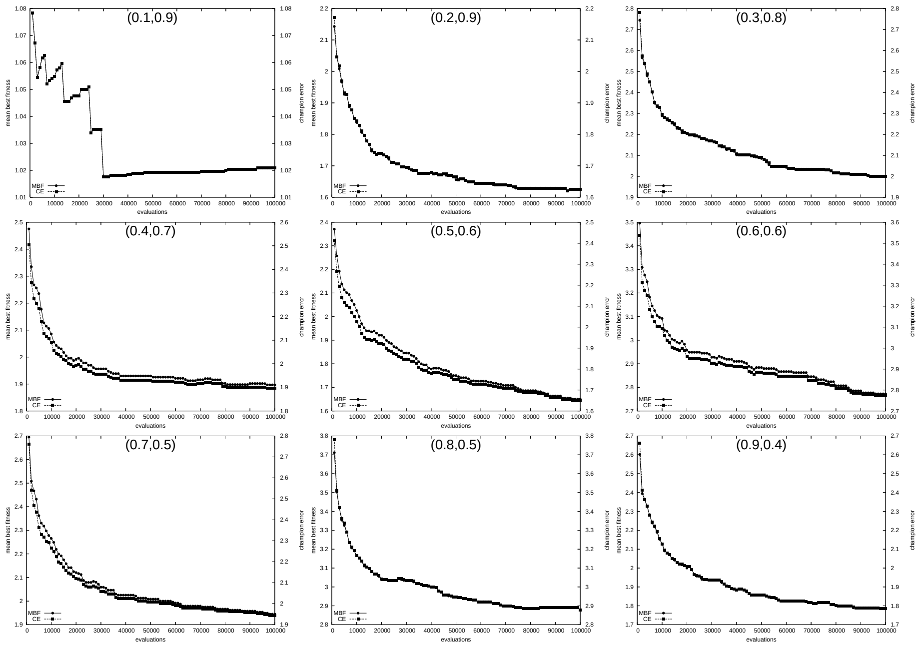

Figure 6.6: *MBF* and *MCE* of the *Intuitive Evolutionary Algorithm*.

6.3 Comparison

Comparing the performance of the algorithms is done in two phases: first a superficial inspection of the results and then a statistical analysis. The comparison focusses primarily on the mushy region because we expect that in the mushy region the differences between the algorithms will be more pronounced.

During the first phase of the comparison we will consider the *SR*, *AES*, and *CC* measures of the algorithms. Table 6.13 shows these measures for the *RSA*, the *HCAWR*, and the *IEA* in the mushy region. The first phase of the comparison is only used to determine which algorithms clearly outperform the others. The best *SR* measure in the table for each density-tightness combination is shown in bold-face. To make the comparison more accurate, we have not rounded the *AES* and *CC* measures in the mushy region. The results in Table 6.13 show that *HCAWR* outperforms all other algorithms when we consider *SR*. The *RSA* has the worst performance of the three algorithms, only for density-tightness combination $(0.1, 0.9)$ does it solve some CSP instances. The *HCAWR* also has the best *AES* of all three algorithms for most density-tightness combinations. Although the *AES* for the *IEA* is sometimes lower, this can be attributed to the inaccuracy of this measure resulting from the lower *SR* that it achieved.

The first phase of the comparison shows that there is a big difference between the performance of the *RSA*, the *HCAWR*, and the *IEA*. It is clear that the *HCAWR* outperforms the other two algorithms. At this point, no further statistical analysis is really necessary to support this conclusion. Not all comparisons will have such a big difference though and we give a method for statistical analysis for use in those cases. We will analyse the performance difference using the two sample t-test over the measures of two algorithms. The standard two sample t-test formulates two hypotheses in order to decide which has the better performance:

$$H_0 : \overline{x}_1 = \overline{x}_2 \tag{6.1}$$

$$H_{a_1} : \overline{x}_1 \neq \overline{x}_2 \tag{6.2}$$

There are two hypotheses, the first one, called the null-hypothesis (H_0), states that the average value of the data points in the first sample is equal to the average value of the data points in the second sample. The second hypothesis, the alternative hypothesis (H_{a_1}), states that the average value of the data points in the first sample is unequal to the average value of the data points in the second sample. The result of the two sample t-test is expressed by a p-value. The p-value gives the probability that the null-hypothesis (6.1) is true and the alternative hypothesis (6.2) is not. The p-value has a range between 0.0 and 1.0, a p-value of 0.5 means that there is an equal probability of both hypotheses being true, signifying that the t-test is inconclusive.

Using hypotheses 6.1 and 6.2 we can determine the probability of two algorithms having equal *SR*, *AES*, or *CC* measures. The data points for the samples are then the values of these measures per run, for a total of 250 data points for each density-tightness combination. Because a run can only be successful or unsuccessful we average these data

$(p_1, \overline{p_2})$	***SR***	***RSA*** ***AES***	***CC***	***SR***	***HCAWR*** ***AES***	***CC***	***SR***	***IEA*** ***AES***	***CC***
$(0.1, 0.9)$	0.532	41146	46214	**1.0**	1858	234242	0.808	4775	23873
$(0.2, 0.9)$	0.0	*undef.*	*undef.*	**1.0**	9712	1267015	0.156	18351	165166
$(0.3, 0.8)$	0.0	*undef.*	*undef.*	**1.0**	15412	2087947	0.064	16009	224123
$(0.4, 0.7)$	0.0	*undef.*	*undef.*	**1.0**	16208	2260634	0.108	19965	359467
$(0.5, 0.6)$	0.0	*undef.*	*undef.*	**1.0**	15487	2237419	0.108	11971	275336
$(0.6, 0.6)$	0.0	*undef.*	*undef.*	**1.0**	18464	2741567	0.068	9929	268078
$(0.7, 0.5)$	0.0	*undef.*	*undef.*	**1.0**	23718	3640630	0.06	19471	623083
$(0.8, 0.5)$	0.0	*undef.*	*undef.*	**1.0**	17290	2722763	0.088	12835	462060
$(0.9, 0.4)$	0.0	*undef.*	*undef.*	**1.0**	15178	2465975	0.088	20571	843407

Table 6.13: Comparison of the *RSA*, the *HCAWR* and the *IEA* in the mushy region.

points per CSP instance for a total number of data points per density-tightness combination of 25. Although this reduces the number of data points, this actually increases the accuracy of the test. The t-test assumes an approximately normal distribution of the data points and, according to the central limit theorem, averaging a sample over a number of sub-sets makes the distribution of the sample approximate the normal distribution.

By altering the alternative hypothesis we can order the algorithms according to performance. Two alternative hypothesis can be used:

$$H_{a_2} : \overline{x}_1 > \overline{x}_2 \quad (6.3)$$

$$H_{a_3} : \overline{x}_1 < \overline{x}_2 \quad (6.4)$$

But as the p-value of alternative hypothesis H_{a_3} (6.4) is equal to one minus the p-value of alternative hypothesis H_{a_2} (6.3), only a single t-test is needed to calculate both probabilities.

The hypotheses used to order the algorithms are:

$$H_0 : \overline{SR}_{A_1} = \overline{SR}_{A_2} \quad (6.5)$$

$$H_{a_1} : \overline{SR}_{A_1} \neq \overline{SR}_{A_2} \quad (6.6)$$

$$H_{a_2} : \overline{SR}_{A_1} > \overline{SR}_{A_2} \quad (6.7)$$

where A_1 is the first algorithms in the test, in this case the *Hill Climber with Restart Algorithm*, and A_2 is the second algorithm in the test, in this case the *Intuitive Evolutionary Algorithm*. The order in which the algorithms are used in the test makes no difference because the p-value of H_{A_3} is one minus the p-value of H_{A_2}.

The p-values for the two alternative hypothesis (the null hypothesis remains the same) are shown in Table 6.14. From the table we can see that the difference between the *SR*

$(\mathbf{p_1}, \overline{\mathbf{p_2}})$	$\mathbf{H_{a_1}}$	$\mathbf{H_{a_2}}$
(0.1,0.9)	0.0	0.0
(0.2,0.9)	0.0	0.0
(0.3,0.8)	0.0	0.0
(0.4,0.7)	0.0	0.0
(0.5,0.6)	0.0	0.0
(0.6,0.6)	0.0	0.0
(0.7,0.5)	0.0	0.0
(0.8,0.5)	0.0	0.0
(0.9,0.4)	0.0	0.0

Table 6.14: Two sample t-Tests of the *HCAWR* and the *IEA*.

of two algorithms is large as the probability for the null hypothesis in both t-tests is 0.0 for all density-tightness combinations. Because all p-values are 0.0 we have shown that the average success rate of *HCAWR* is not equal to the average success rate of *IEA* but that it is in fact larger. The probability that it is not so is in fact 0.0. Clearly, the *Hill Climber with Restart Algorithm* outperforms the *Intuitive Evolutionary Algorithm* and the *Random Search Algorithm*.

Chapter 7

Evolutionary Algorithms for Solving the Constraint Satisfaction Problem

This chapter gives a inventory of evolutionary algorithms for solving constraint satisfaction problems. The algorithms included cover the different types of methods used in evolutionary algorithms for solving constraint satisfaction problems. Each algorithm is discussed in its own section and included are a full description of the algorithm, a specification of the characteristics of the algorithm, the parameter setup used for the experiments and an overview of the results of these experiments. A comparison of the performance of the algorithms is given in the next chapter.

7.1 *Heuristic Evolutionary Algorithm*

In [28, 29], A.E. Eiben *et al.* propose to incorporate existing heuristics for the constraint satisfaction problem into the genetic operators of evolutionary algorithms.
These heuristics are used as *rules-of-thumb* to guide the operators to choose which variables or values to change. The heuristics are divided into two categories:

Variable Heuristics A variable heuristic chooses which variable the operator should re-label. The most commonly used variable heuristic for the constraint satisfaction problem chooses the variable with the largest number of relevant violated constraints for a particular candidate solution. By re-labelling this variable, the biggest improvement by a single re-labelling can be made.

Value Heuristics A value heuristic chooses which value a chosen variable should be re-labelled with. The most commonly used value heuristic for the constraint satisfaction problem chooses the value which satisfies the most relevant constraints.

This heuristic was also used in the *Hill Climber with Restart Algorithm*.

Experiments with the *Hill Climber with Restart Algorithm* showed that the exclusive use of heuristics leads to a convergence on a local optimum of the population when the neighbourhood of a series of candidate solutions is explored exhaustively. This prevents the algorithm from reaching the global optimum and in the *Hill Climber with Restart Algorithm* a restart strategy is used to counter this behaviour. Although a restart strategy is also possible for evolutionary algorithms, more commonly, the mutation operator is used for this. Heuristics are then incorporated in the crossover operator only. In [28, 29], A.E. Eiben *et al.*, identified two ways of incorporating heuristics into a recombination operator:

The Asexual Heuristic Operator This operator uses both the variable and the value heuristic. First it uses the variable heuristic to select a number of variables. These variables are then re-labelled with a value chosen by the value heuristic. Variables are re-labelled iteratively, taking the effects of previous re-labellings into account. The number of variables to re-label is determined by a parameter of the operator. In [18], it was found that selecting one quarter of the variables has the best overall performance for the constraint satisfaction problem. The asexual operator produces one child for each parent and can be used both as a crossover and a mutation operator.

The Multi-Parent Heuristic Operator The multi-parent heuristic operator uses the multi-parent crossover mechanism of scanning. The scanning mechanism determines the values of the children by scanning the values of the parents for each variable. The multi-parent heuristic operator creates one child from more than two parents. The number of parents is determined by a parameter of the operator. In [18], it was found that using 5 parents produced the best overall performance. No variable heuristic is used in the multi-parent heuristic operator since the scanning mechanism considers all variables. The value heuristic is used to select the value for each variable of the child. Only the values of the parents are considered.

Two versions of the *Heuristic Evolutionary Algorithm* (*HeuristicEA*) are defined, one for each heuristic operator. In [18], another, third, version was defined, using the multi-parent heuristic operator as a crossover operator and the asexual heuristic operator as a mutation operator. In the same paper, a fourth version, using the asexual heuristic operator as both a crossover and a mutation operator was rejected, because it would simply entail a double application of the same operator. The three versions of the *Heuristic Evolutionary Algorithm* are abbreviated as:

HEA1 using the asexual heuristic operator as a crossover operator;

HEA2 using the multi-parent heuristic operator as a crossover operator; and

HEA3 using the multi-parent heuristic operator as a crossover operator and the asexual heuristic operator as a mutation operator.

7.1.1 *HeuristicEA* Characteristics and Parameter Setup

Tables 7.1, 7.3, and 7.5 show the characteristics tables of the *HEA1*, the *HEA2*, and the *HEA3* respectively. All three versions of the *Heuristic Evolutionary Algorithm* use a steady state evolutionary model, an ordered set of values representation, fitness function f_1, a biased ranking parent selection operator, and a replace worst survivor selection operator. These characteristics are explained in Chapter 5. The *HEA1* and the *HEA2* use a uniform random mutation operator. The three versions of the *Heuristic Evolutionary Algorithm* use the heuristic operators as explained in the previous section.

Tables 7.2, 7.4, and 7.6 show the parameter tables of the *HEA1*, the *HEA2*, and the *HEA3*. All three versions of the *Heuristic Evolutionary Algorithm* have a population of 10 individuals (Population Size), from which 10 parents are selected (Selection Size) using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias). The crossover operator of all three versions is applied with a crossover rate of 1.0 (Crossover Rate) and the uniform random mutation operator in the *HEA1* and the *HEA2* uses a mutation rate of 0.1 (Mutation Rate). A mutation rate of 0.1 here means that there is a 0.1 probability of re-labelling a variable where each variable in the individual is checked. The experiments of all three versions of the *Heuristic Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations). The asexual heuristic operator of the *HEA1* and the *HEA3* changes one quarter of the ten variables of the CSP instances in our test-set, rounded upwards to 3 (Change Number of Variables). The multi-parent heuristic operator uses 5 parents (Number of Parents).

7.1.2 *HeuristicEA* Experimental Results

Tables 7.7, 7.10, and 7.13, show that both the *HEA1* and the *HEA3* solve the CSP instances in the solvable region in almost all runs. In the mushy region itself, both the *HEA1* and the *HEA3* have a *SR* of 1.0 for density-tightness combination $(0.1, 0.9)$. The *HEA2* has the worst *SR* throughout the density-tightness combinations in the mushy region, in general solving the CSP instances there in only a few runs. Tables 7.8, 7.11, and 7.14 show that relative to the *Intuitive Evolutionary Algorithm*, the *HEA1* and the *HEA2* use a low *AES* in the mushy region. Only the *HEA3* uses a high *AES* in the mushy region. On the other hand, Tables 7.9, 7.12, and 7.15 show that all three versions of the *Heuristic Evolutionary Algorithm* use a high *CC* in the mushy region. The high *CC* are used by the heuristic operators. The heuristics use the conflict checks to determine which variable or value to choose. As these heuristics are used outside the objective function, this is not reflected in a high *AES*.

The *UIC* plots of all three versions of the *Heuristic Evolutionary Algorithm* in Figures 7.1, 7.3, and 7.5 all show that throughout the run, all versions keep evaluating new unique individuals. Of the three versions, *HEA1* searches through the largest portion of the search space and, on average, is the least close to a premature convergence to a local optimum at the end of its runs. The runs for both the *HEA1* and the *HEA3* solved all CSP instances in density-tightness combination (0.1,0.9) before the second

HEA1	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	f_1
Crossover operator	Asexual Heuristic
Mutation operator	Uniform Random Mutation
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	None

Table 7.1: Characteristics of the *HEA1*.

HEA1	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Change Number of Variables	3
Ranking Bias	1.5
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.2: Parameters of the *HEA1*.

HEA2	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	f_1
Crossover operator	Multi-Parent Heuristic
Mutation operator	Uniform Random Mutation
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	None

Table 7.3: Characteristics of the *HEA2*.

HEA2	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Number of Parents	5
Ranking Bias	1.5
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.4: Parameters of the *HEA2*.

HEA3	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	f_1
Crossover operator	Multi-Parent Heuristic
Mutation operator	Asexual Heuristic
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	None

Table 7.5: Characteristics of the *HEA3*.

HEA3	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Number of Parents	5
Change Number of Variables	3
Ranking Bias	1.5
Crossover Rate	1.0

Table 7.6: Parameters of the *HEA3*.

interval, i.e., before 2000 evaluations. The *UIC* plots for these two algorithms therefore show only a single dot. The *UIC* plots for the *HEA2* and the *HEA3* show that these two algorithms search through the smallest portion of the search space and that, on average, by the end of their runs, their populations have almost converged on a local optimum. Both the *HEA2* and the *HEA3* use the multi-parent heuristic operator and the *UIC* plots suggest that this operator limits the amount of search space that is searched.

The *MBF/MCE* plots of all three versions of the *Heuristic Evolutionary Algorithm* in Figures 7.2, 7.4, and 7.6 show that, on average, the *MBF* is close to the *MCE*. The reason for this is that the f_1 objective function is the same as the *MCE* measure. The difference between the *MBF* and the *MCE* in the *HEA1* and the *HEA3* can be explained by the influence of finding a solution has on these measures. Whereas the *MBF* is calculated by averaging over the best fitness values of the individuals in the population, the *MCE* is calculated over a single value at the same interval. Neither measure is calculated over runs that are not yet successful but as more runs end successfully, the average of both measures is calculated over fewer runs. For the *HEA1* and the *HEA3*, which have more successful runs, this is shown as a less regular plot than for the *HEA2*, which has fewer successful runs.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.928	0.504
0.5	1.0	1.0	1.0	1.0	1.0	0.872	0.4	0.428	—
0.6	1.0	1.0	1.0	0.98	0.504	0.42	—	—	—
0.7	1.0	1.0	0.888	0.572	—	—	—	—	—
0.8	1.0	1.0	0.556	—	—	—	—	—	—
0.9	1.0	0.892	—	—	—	—	—	—	—

Table 7.7: *SR* of the *HEA1*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	12	14	16	17	19	19
0.2	10	12	16	18	19	20	21	23	26
0.3	12	17	19	20	23	26	31	35	44
0.4	14	19	22	25	33	42	69	7980	2789
0.5	18	20	27	37	102	13528	1951	7603	—
0.6	19	23	40	2089	3387	5704	—	—	—
0.7	20	33	11548	1448	—	—	—	—	—
0.8	24	53	3931	—	—	—	—	—	—
0.9	37	335	—	—	—	—	—	—	—

Table 7.8: *AES* of the *HEA1*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	129	546	1127	2139	3504	4171	4996	5120
0.2	172	1263	3168	4486	5178	5635	6388	7199	8874
0.3	813	3563	4984	5636	7198	8745	11319	14059	18864
0.4	2022	4633	6097	7891	12080	16824	31613	444585	$2 \cdot 10^6$
0.5	3944	5311	8780	14073	47751	435723	$1 \cdot 10^6$	$4 \cdot 10^6$	—
0.6	4635	6865	15594	41547	$1 \cdot 10^6$	$3 \cdot 10^6$	—	—	—
0.7	5034	11831	197501	760728	—	—	—	—	—
0.8	6993	22073	$2 \cdot 10^6$	—	—	—	—	—	—
0.9	13715	166541	—	—	—	—	—	—	—

Table 7.9: *CC* of the *HEA1*.

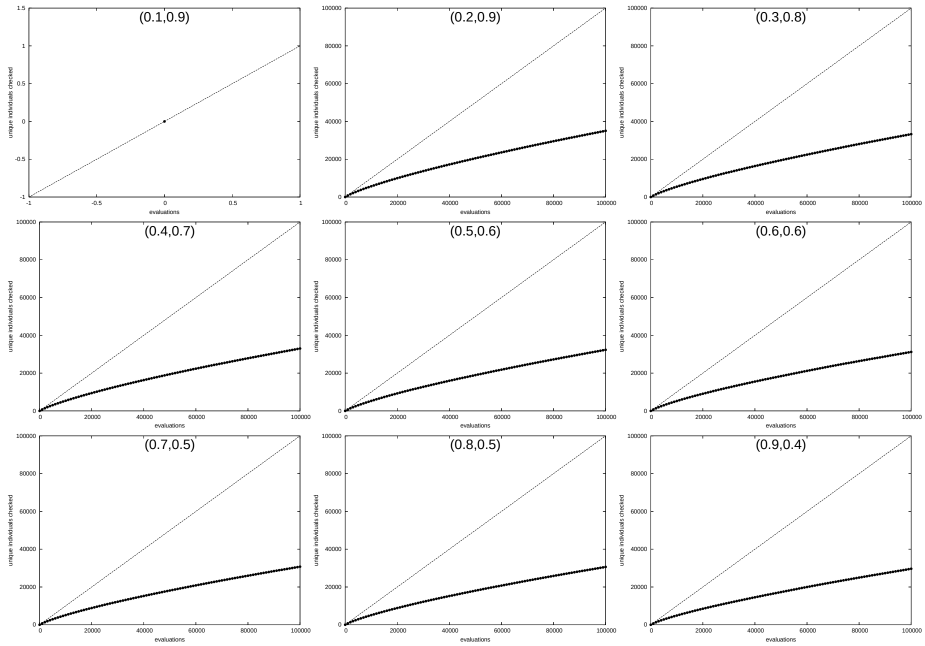

Figure 7.1: *UIC* of the *HEA1*.

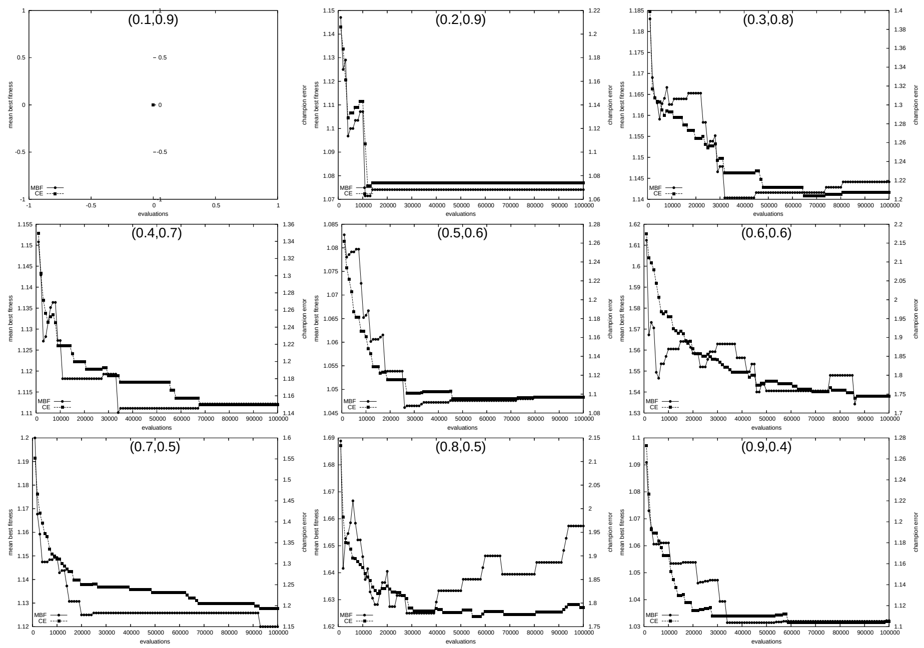

Figure 7.2: *MBF* and *MCE* of the *HEA1*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	0.988	0.948	0.808
0.4	1.0	1.0	1.0	1.0	0.98	0.888	0.572	0.288	0.076
0.5	1.0	1.0	1.0	0.92	0.592	0.232	0.04	0.064	—
0.6	1.0	1.0	0.832	0.444	0.072	0.056	—	—	—
0.7	1.0	0.904	0.324	0.08	—	—	—	—	—
0.8	0.956	0.616	0.068	—	—	—	—	—	—
0.9	0.764	0.188	—	—	—	—	—	—	—

Table 7.10: *SR* of the *HEA2*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	12	14	18	21	25	27
0.2	10	12	19	24	34	43	58	86	193
0.3	12	19	33	59	88	183	1590	4047	5461
0.4	15	29	73	182	2281	5448	11402	11387	16609
0.5	22	58	1171	5280	9081	14371	14444	13596	—
0.6	35	391	4589	16208	10727	13596	—	—	—
0.7	134	5287	12545	21876	—	—	—	—	—
0.8	2791	8732	13660	—	—	—	—	—	—
0.9	5862	14268	—	—	—	—	—	—	—

Table 7.11: *AES* of the *HEA2*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	145	691	1511	3273	6054	8125	10826	12754
0.2	213	1704	6203	9965	16787	23099	34199	54168	131724
0.3	1111	6547	15739	34054	54646	121997	$1 \cdot 10^6$	$3 \cdot 10^6$	$4 \cdot 10^6$
0.4	3232	12922	43767	119653	$2 \cdot 10^6$	$4 \cdot 10^6$	$8 \cdot 10^6$	$8 \cdot 10^6$	$12 \cdot 10^6$
0.5	8319	33141	800097	$4 \cdot 10^6$	$6 \cdot 10^6$	$10 \cdot 10^6$	$10 \cdot 10^6$	$8 \cdot 10^6$	—
0.6	16996	260913	$3 \cdot 10^6$	$11 \cdot 10^6$	$7 \cdot 10^6$	$10 \cdot 10^6$	—	—	—
0.7	84533	$4 \cdot 10^6$	$9 \cdot 10^6$	$15 \cdot 10^6$	—	—	—	—	—
0.8	$2 \cdot 10^6$	$6 \cdot 10^6$	$9 \cdot 10^6$	—	—	—	—	—	—
0.9	$4 \cdot 10^6$	$10 \cdot 10^6$	—	—	—	—	—	—	—

Table 7.12: *CC* of the *HEA2*.

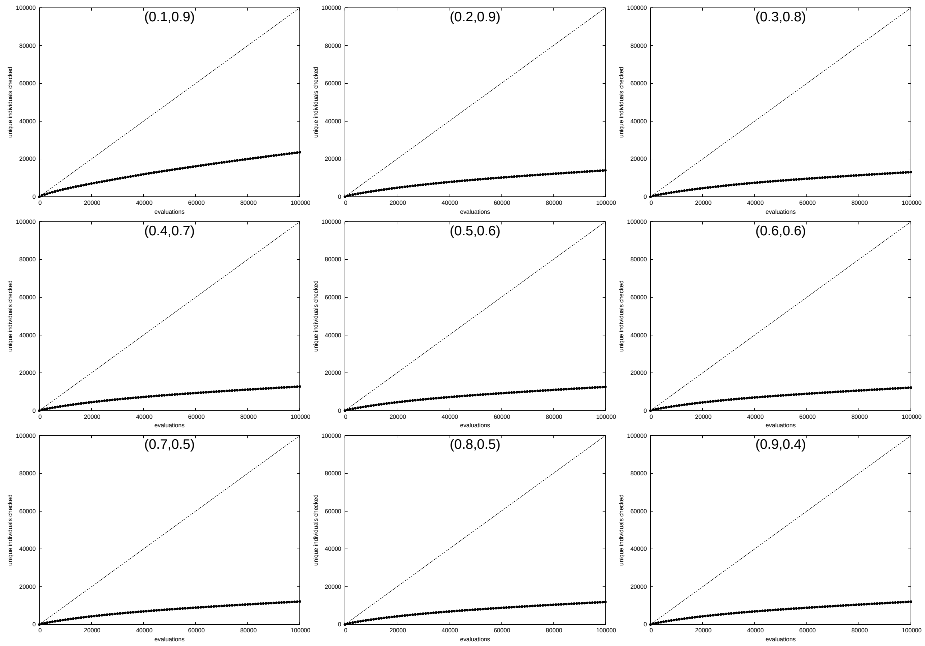

Figure 7.3: *UIC* of the *HEA2*.

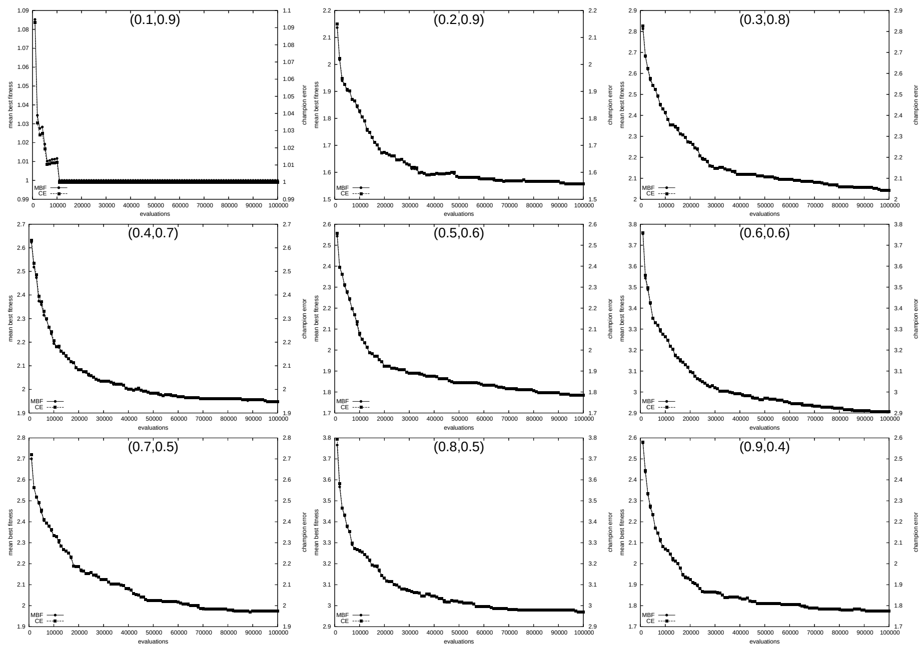

Figure 7.4: *MBF* and *MCE* of the *HEA2*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.992	0.76
0.5	1.0	1.0	1.0	1.0	1.0	0.968	0.588	0.488	—
0.6	1.0	1.0	1.0	1.0	0.692	0.44	—	—	—
0.7	1.0	1.0	0.976	0.712	—	—	—	—	—
0.8	1.0	1.0	0.688	—	—	—	—	—	—
0.9	1.0	0.984	—	—	—	—	—	—	—

Table 7.13: *SR* of the *HEA3*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	12	14	16	17	19	19
0.2	10	12	16	18	19	20	20	20	20
0.3	12	17	19	20	20	20	20	21	24
0.4	14	19	20	20	20	23	33	339	1563
0.5	18	20	20	22	32	438	969	1258	—
0.6	19	20	22	47	2382	988	—	—	—
0.7	20	21	432	1404	—	—	—	—	—
0.8	20	31	1635	—	—	—	—	—	—
0.9	26	419	—	—	—	—	—	—	—

Table 7.14: *AES* of the *HEA3*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	212	1359	2983	5858	9723	11523	13749	13904
0.2	412	3602	9091	12934	14486	15237	15643	15792	15887
0.3	2347	10526	14551	15027	15467	15583	15778	17054	21967
0.4	6032	13443	15329	15357	16137	20332	35849	509570	$2 \cdot 10^6$
0.5	11885	14905	15478	18130	34542	660473	$1 \cdot 10^6$	$2 \cdot 10^6$	—
0.6	13827	15246	18087	55841	$4 \cdot 10^6$	$1 \cdot 10^6$	—	—	—
0.7	14679	17182	630507	$2 \cdot 10^6$	—	—	—	—	—
0.8	15607	32547	$2 \cdot 10^6$	—	—	—	—	—	—
0.9	23899	621391	—	—	—	—	—	—	—

Table 7.15: *CC* of the *HEA3*.

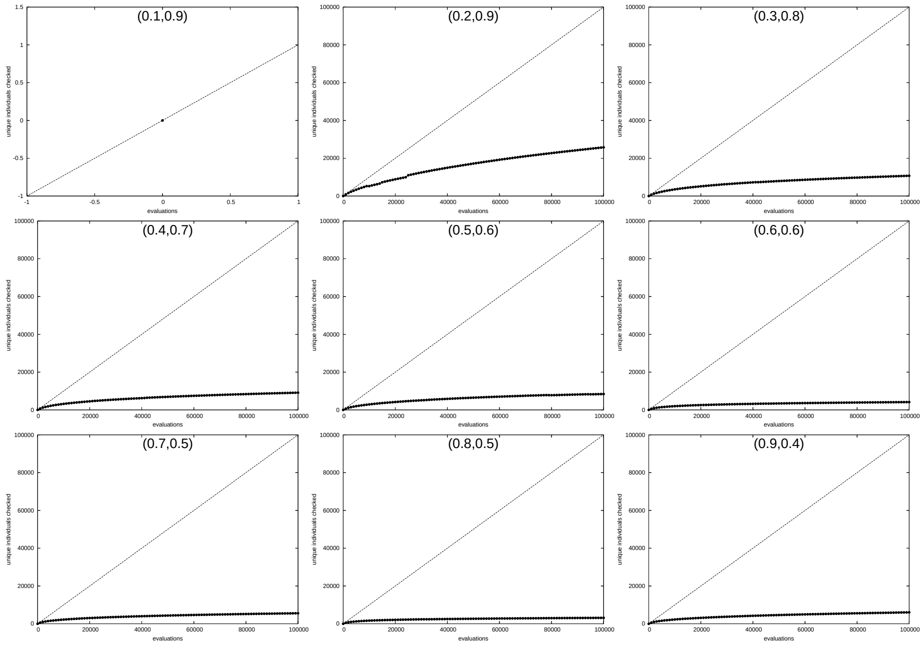

Figure 7.5: *UIC* of the *HEA3*.

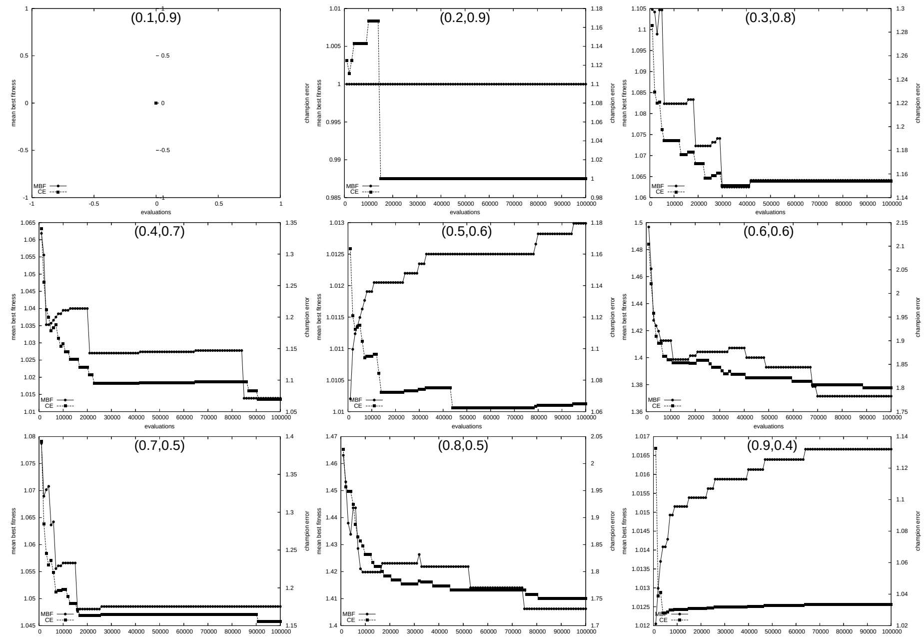

Figure 7.6: *MBF* and *MCE* of the *HEA3*.

7.2 *Arc Evolutionary Algorithm*

The *Arc Evolutionary Algorithm* (*ArcEA*) was first introduced in [74] by M.-C. Riff-Rojas. Based on *HEA1*, in addition the *ArcEA* uses constraint network information in the objective function of the algorithm. In [75], *ArcEA* was further adapted by replacing the asexual heuristic operator with a special crossover operator using information gathered by the objective function. In [76], in a third and final version of the *ArcEA*, the crossover operator was made more adaptive. In addition, the uniform random mutation operator used by the first version was replaced by a mutation operator also using constraint network information. All three versions of the *ArcEA* used a specially designed parent selection operator. In total, five new parts were introduced:

The Arc Objective Function This objective function takes its name from the definition of an arc in the constraint satisfaction problem. An (second order) arc is three variables and their two relevant constraints. The arc objective function uses constraint network information by calculating the *error evaluation* for each constraint in the problem. The error evaluation of a constraint is defined as follows: for a binary CSP $\langle X, C, D\rangle$, two variables $x_1 \in X$ and $x_2 \in X$, $x_1 \neq x_2$, both relevant to constraint $c \in C$, are also relevant to the constraints in $C_1 \subset C$ and $C_2 \subset C$ respectively; the error evaluation of c is then the size of the subset $C' \subset C$, where $C' = C_1 \cap C_2$. Because the constraint network of a CSP remains static, the error evaluation of all constraints can be calculated at initialisation of the algorithm. The arc objective function calculates the fitness value of an individual by adding the error evaluation of all violated constraints in the candidate solution of the individual. Constraints with a high error evaluation are relevant by arc to more variables and are thus harder to satisfy. By focussing on these constraints, the arc objective function directs the search of the evolutionary algorithm towards solving these constraints first.

The Arc Crossover Operator The arc crossover operator constructs a single child from two parents. The construction starts with a child in which none of the variables are labelled. The variables in the child are then labelled iteratively considering each constraint in the CSP in random order using the labels of the parents. The constraint currently considered is denoted by c and the two relevant variables to c are denoted by x_1 and x_2. The following three cases can then be distinguished:

1. Both variables are unlabelled in the child. Three cases are possible:
   (a) The compound label with variable set $S = \{x_1, x_2\}$ of neither parent satisfies c. The compound label that minimises the summed error evaluation of the constraints relevant to x_1 or x_2 whose other relevant variable is already labelled in the child is used to label x_1 and x_2 in the child.
   (b) The compound label with variable set $S = \{x_1, x_2\}$ of exactly one parent satisfies c. That compound label is used to label x_1 and x_2 in the child.

(c) The compound labels with variable set $S = \{x_1, x_2\}$ of both parents satisfies c. The compound label from the parent with the best fitness value is used to label x_1 and x_2 in the child.

2. One variable is unlabelled in the child. The label in the two parents that minimises the summed error evaluation of the constraints relevant to the unlabelled variable is used to label the unlabelled variable in the child.
3. Both variables are labelled in the child. Nothing is done and the next constraint is considered.

When the summed error evaluation of the constraints relevant to two variables are tied, the value used is determined randomly. A variable relevant to any constraint in the CSP is labelled by a random value from its domain.

The Constraint Dynamic Adaptive Crossover Operator This operator uses the same construction method as the arc crossover operator but replaces the random order in which the constraints are considered with an adaptive ordering based on the error evaluation of the constraints in both parents. The ordering is divided into three parts: first the constraints that are violated in both parents are considered, then the constraints that are violated in one of the parents are considered, finally, constraints that are not violated in both parents are considered. In each of these parts the constraints are ordered based on their error evaluation: constraints with a higher error evaluation are considered before constraints with a lower error evaluation. By using this ordering, the constraint dynamic adaptive crossover operator focusses on constraints that have not yet been satisfied before constraints that have already be satisfied. The operator is dynamic because it changes focus based on the parent pair it is supplied with. Focus also changes during the run of the algorithm.

The Arc Mutation Operator The arc mutation operator also uses the error evaluation of constraints. First it selects a variable to re-label uniform randomly. It then re-labels this variable with the value that minimises the summed error evaluation of the constraints relevant to the selected variable.

The α-β Parent Selection Operator The α-β parent selection operator splits the population into three groups. The first group includes all individuals with a fitness value better than the mean fitness value of the population. The second group includes all individuals with a fitness value better than the mean plus the standard deviation of the fitness values. If the fitness function is to be maximised, the standard deviation is subtracted. The third group then includes all remaining individuals in the population. The operator then selects individuals proportionally from these three groups depending on the α and β parameters of the operator. If both α and β are given as percentages, α percent of the selection size are selected from the first group, $\beta - \alpha$ percent are selected from the second group and $100\% - \beta$ percent are selected from the third group. Selection from within a group is done uniform randomly and with repetition. Commonly used parameters are $\alpha = 50\%$ and $\beta = 85\%$. Note that the α-β parent selection

ArcEA1	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	Arc Objective Function
Crossover operator	Asexual Heuristic
Mutation operator	Uniform Random Mutation
Parent Selection	α-β Parent Selection
Survivor Selection	Replace Worst
Other Functions	None

Table 7.16: Characteristics of the *ArcEA1*.

ArcEA1	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Change Number of Variables	3
Selection α	0.5
Selection β	0.85
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.17: Parameters of the *ArcEA1*.

operator is similar to a linear ranking parent selection operator in which there are only three ranks where parents are selected from these ranks with a fixed probability (determined by α and β).

The three papers of M.-C. Riff-Rojas ([74, 75, 76]) define three different evolutionary algorithms. The three algorithms will be abbreviated by: *ArcEA1*, *ArcEA2*, and *ArcEA3*. *ArcEA1* is an adaptation of *HEA1* with the objective function replaced by the arc objective function and the biased ranked parent selection operator by the arc parent selection operator. *ArcEA2* then replaces the asexual heuristic operator in *ArcEA1* with the arc crossover operator and the uniform random mutation operator with the arc mutation operator. *ArcEA3* then replaces the arc crossover operator of the *ArcEA2* with the constraint dynamic crossover operator.

7.2.1 *ArcEA* Characteristics and Parameter Setup

Tables 7.16, 7.18, and 7.20 show the characteristics tables of the *ArcEA1*, the *ArcEA2*, and the *ArcEA3* respectively. All three versions of the *Arc Evolutionary Algorithm* use a steady state evolutionary model, an ordered set of values representation, and a replace worst survivor selection operator, all explained in Chapter 5. The other characteristics of the three versions of the *Arc Evolutionary Algorithm* were given in the previous

ArcEA2	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	Arc Objective Function
Crossover operator	Arc Crossover
Mutation operator	Arc Mutation
Parent Selection	α-β Parent Selection
Survivor Selection	Replace Worst
Other Functions	None

Table 7.18: Characteristics of the *ArcEA2*.

ArcEA2	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Selection α	0.5
Selection β	0.85
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.19: Parameters of the *ArcEA2*.

section.

Tables 7.17, 7.19, and 7.21 show the parameter tables of the *ArcEA1*, the *ArcEA2*, and the *ArcEA3*. All three versions of the *Arc Evolutionary Algorithm* have a population of 10 individuals (Population Size), from which 10 parents are selected (Selection Size) using the α-β parent selection operator with an α of 0.5 (Selection α) and a β of 0.85 (Selection β). The crossover operator of all three versions is applied with a crossover rate of 1.0 (Crossover Rate) and the mutation operator is applied with a mutation rate of 0.1 (Mutation Rate). The experiments of all three versions of the *Arc Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations). The asexual heuristic operator of *ArcEA1* changes 3 variables in the individual (Change Number of Variables).

7.2.2 *ArcEA* Experimental Results

Tables 7.22, 7.25, and 7.28, show that the *ArcEA1* has the highest *SR* of the three versions of the *Arc Evolutionary Algorithm*. Both the *ArcEA2* and the *ArcEA3* do not solve the CSP instances in the mushy region as often as the *ArcEA1* does. This suggests that the addition of the arc crossover operator and the constraint dynamic adaptive crossover operator does not contribute to a high *SR*. Tables 7.23, 7.26, and 7.29 show that the *AES* of all three versions of the *Arc Evolutionary Algorithm* is relatively low.

ArcEA3	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	Arc Objective Function
Crossover operator	Constraint Dynamic Adaptive Crossover
Mutation operator	Arc Mutation
Parent Selection	α-β Parent Selection
Survivor Selection	Replace Worst
Other Functions	None

Table 7.20: Characteristics of the *ArcEA3*.

ArcEA3	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Selection α	0.5
Selection β	0.85
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.21: Parameters of the *ArcEA3*.

However, because the *SR* of the *ArcEA2* and the *ArcEA3* are not as high as the *SR* of the *ArcEA1*, these *AES* values are less accurate. This is because the *AES* (as the *CC*) is calculated over successful runs only and with less successful runs, the accuracy of the *AES* measures is reduced. The same is seen for the *CC* measure in Tables 7.24, 7.27, and 7.30.

The *UIC* plots of all three versions of the *Arc Evolutionary Algorithm* in Figures 7.7, 7.9, and 7.11 show that both the *ArcEA2* and the *ArcEA3* search only a limited portion of the search space. These plots also show that after only a few evaluations, almost no new unique individuals are evaluated, suggesting premature convergence of the population. The *ArcEA1*, much like the *HEA1*, searches through a larger portion of the search space and shows no sign of premature convergence of the population. The *MBF/MCE* plots in Figures 7.8, 7.10, and 7.12 show little difference between how the arc objective function calculates fitness values and the *MCE*. Although the arc objective function uses constraint network information, this did not give the algorithm an edge over, for example, the *HEA1*. One has to keep in mind that the *Arc Evolutionary Algorithm* was written with CSPs with varying tightness is mind whereas in the test-set we use all constraints have approximately the same tightness. With no hard to satisfy constraints to focus on, the direction provided by the more elaborate arc objective function does not result in a better *SR*. The same (but less clear from the experiments we ran) can probably be said for the other components of the *Arc Evolutionary Algorithm* that use the error evaluation of the constraints. We expect that on a test-set with CSP instances with more variance between the tightness of constraints, the use of error evaluations would give an edge to the *Arc Evolutionary Algorithm*. For all three versions of the *Arc Evolutionary Algorithm*, the *MBF* and the *MCE* are close together and almost completely monotonic in their decrease. The *MBF/MCE* plots show no sign of premature convergence of the population. The *UIC* and *MBF/MCE* plots together do not point to premature convergence of the population as the reason for the low *SR* of the *ArcEA2* and the *ArcEA3*, but, instead, point to a lack of effectiveness of the algorithms to find solutions within the number of evaluations allowed. The *MBF/MCE* plots are fairly regular for the *ArcEA2* and the *ArcEA3* because of the low number of successful runs over which the measures were calculated.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	0.968	0.704	0.3
0.5	1.0	1.0	1.0	1.0	0.936	0.644	0.22	0.24	—
0.6	1.0	1.0	0.996	0.884	0.312	0.284	—	—	—
0.7	1.0	1.0	0.684	0.384	—	—	—	—	—
0.8	1.0	0.948	0.368	—	—	—	—	—	—
0.9	0.988	0.688	—	—	—	—	—	—	—

Table 7.22: *SR* of the *ArcEA1*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	12	14	16	17	19	20
0.2	10	12	16	19	21	24	27	29	33
0.3	12	17	21	25	29	33	40	81	156
0.4	14	20	27	33	44	89	297	2815	5067
0.5	18	24	34	104	287	2732	2116	778	—
0.6	20	30	112	653	962	2099	—	—	—
0.7	23	45	1561	4403	—	—	—	—	—
0.8	71	398	2008	—	—	—	—	—	—
0.9	279	3467	—	—	—	—	—	—	—

Table 7.23: *AES* of the *ArcEA1*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	110	345	661	1220	2005	2381	2971	3251
0.2	111	688	1751	2587	3309	4129	5076	5877	7023
0.3	436	1863	3079	4118	5530	6819	8942	20581	43490
0.4	1036	2641	4645	6297	9516	21302	79078	794981	$2 \cdot 10^6$
0.5	2038	3713	6635	25542	74370	765289	568997	220251	—
0.6	2590	5310	29708	174661	260864	588314	—	—	—
0.7	3448	9238	412297	$1 \cdot 10^6$	—	—	—	—	—
0.8	15947	102493	523101	—	—	—	—	—	—
0.9	67462	865715	—	—	—	—	—	—	—

Table 7.24: *CC* of the *ArcEA1*.

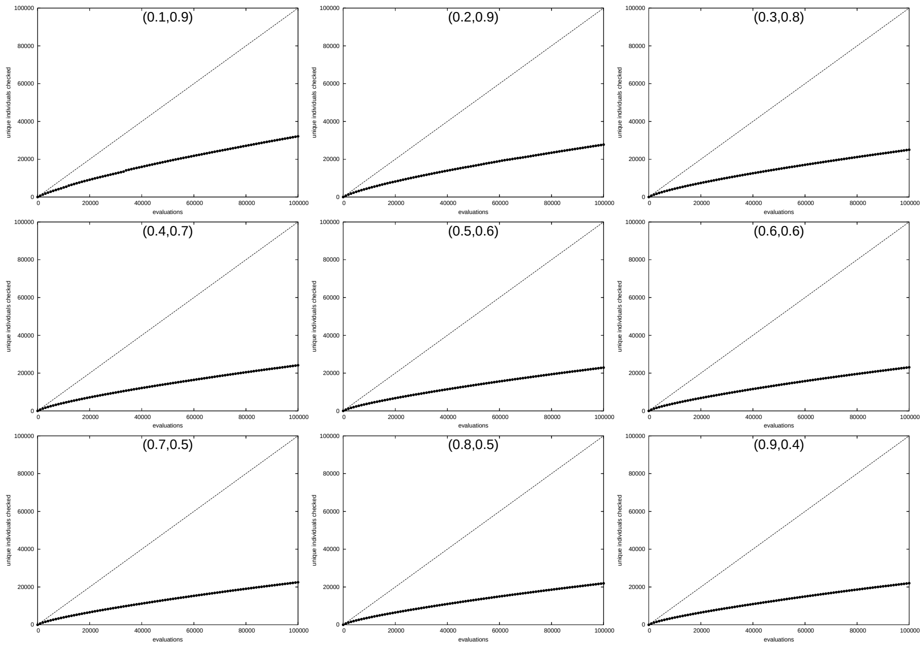

Figure 7.7: *UIC* of the *ArcEA1*.

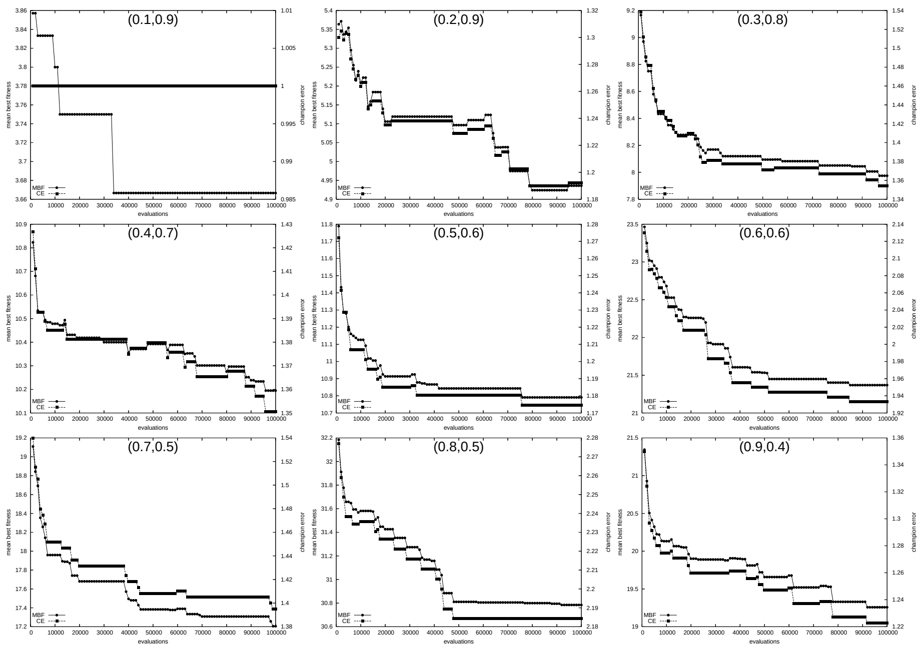

Figure 7.8: *MBF* and *MCE* of the *ArcEA1*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.996
0.3	1.0	1.0	1.0	1.0	0.996	0.972	0.876	0.756	0.456
0.4	1.0	1.0	0.992	0.968	0.84	0.556	0.224	0.108	0.012
0.5	1.0	0.988	0.932	0.732	0.252	0.1	0.008	0.008	—
0.6	1.0	0.948	0.628	0.208	0.016	0.024	—	—	—
0.7	0.984	0.712	0.168	0.02	—	—	—	—	—
0.8	0.94	0.408	0.016	—	—	—	—	—	—
0.9	0.708	0.12	—	—	—	—	—	—	—

Table 7.25: *SR* of the *ArcEA2*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	13	16	22	26	31	33
0.2	10	13	22	30	42	51	72	84	188
0.3	12	24	40	55	163	565	1098	1565	2372
0.4	16	37	71	478	1712	1728	2667	396	250
0.5	28	55	338	1509	2208	1044	218	1953	—
0.6	37	237	2184	1720	494	186	—	—	—
0.7	164	2029	494	186	—	—	—	—	—
0.8	1544	1747	362	—	—	—	—	—	—
0.9	2804	8269	—	—	—	—	—	—	—

Table 7.26: *AES* of the *ArcEA2*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	95	209	424	957	1863	2883	4170	5219
0.2	56	240	1008	2043	3953	5751	10128	13408	35851
0.3	102	710	2253	4176	17224	72339	167911	269468	466275
0.4	212	1306	4340	40796	188529	223412	409633	67419	48171
0.5	509	2101	22263	129995	242882	134956	40616	51267	—
0.6	751	10262	147895	148207	23545	253660	—	—	—
0.7	3814	87257	32826	15292	—	—	—	—	—
0.8	38036	74987	24073	—	—	—	—	—	—
0.9	68308	351511	—	—	—	—	—	—	—

Table 7.27: *CC* of the *ArcEA2*.

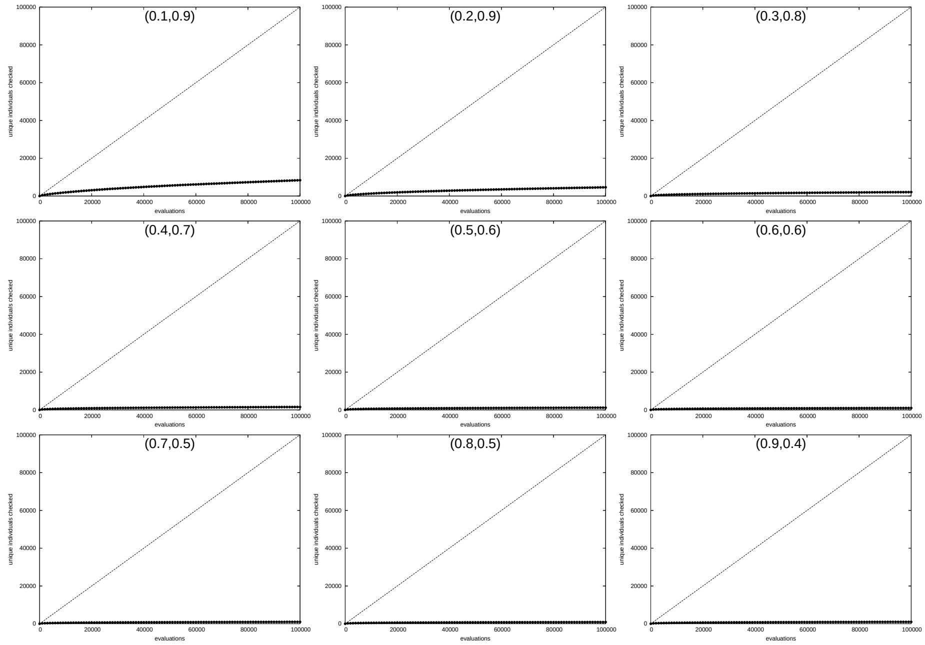

Figure 7.9: *UIC* of the *ArcEA2*.

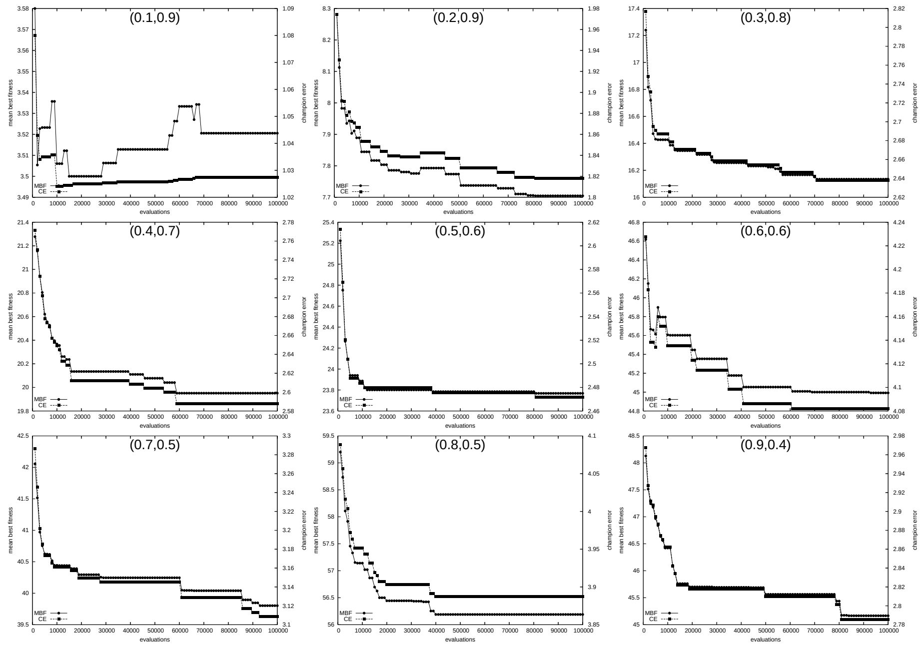

Figure 7.10: *MBF* and *MCE* of the *ArcEA2*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.988
0.3	1.0	1.0	1.0	1.0	0.992	0.988	0.888	0.752	0.532
0.4	1.0	1.0	1.0	1.0	0.868	0.564	0.272	0.1	0.008
0.5	1.0	0.996	0.912	0.724	0.248	0.108	0.012	0.004	—
0.6	1.0	0.944	0.656	0.2	0.012	0.028	—	—	—
0.7	0.976	0.696	0.196	0.032	—	—	—	—	—
0.8	0.908	0.356	0.024	—	—	—	—	—	—
0.9	0.692	0.128	—	—	—	—	—	—	—

Table 7.28: *SR* of the *ArcEA3*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	13	16	21	25	31	32
0.2	10	13	22	30	39	52	64	79	196
0.3	12	24	42	57	91	363	1132	1920	2799
0.4	17	37	71	452	426	2482	2467	5444	1225
0.5	28	66	767	775	1225	413	2720	290	—
0.6	41	360	995	574	173	8060	—	—	—
0.7	81	581	2605	2906	—	—	—	—	—
0.8	250	2991	648	—	—	—	—	—	—
0.9	2036	4056	—	—	—	—	—	—	—

Table 7.29: *AES* of the *ArcEA3*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	97	236	520	1244	2387	3519	5477	6506
0.2	58	311	1294	2582	4686	7808	11837	16649	50139
0.3	112	989	3169	5877	12605	63628	229993	453099	760467
0.4	296	1771	5897	52190	63303	429485	507992	$1 \cdot 10^6$	333482
0.5	682	3603	69641	90344	177084	72220	596776	66792	—
0.6	1194	21573	92708	68401	25600	$1 \cdot 10^6$	—	—	—
0.7	2589	35258	244572	353944	—	—	—	—	—
0.8	8657	177186	60408	—	—	—	—	—	—
0.9	72839	251766	—	—	—	—	—	—	—

Table 7.30: *CC* of the *ArcEA3*.

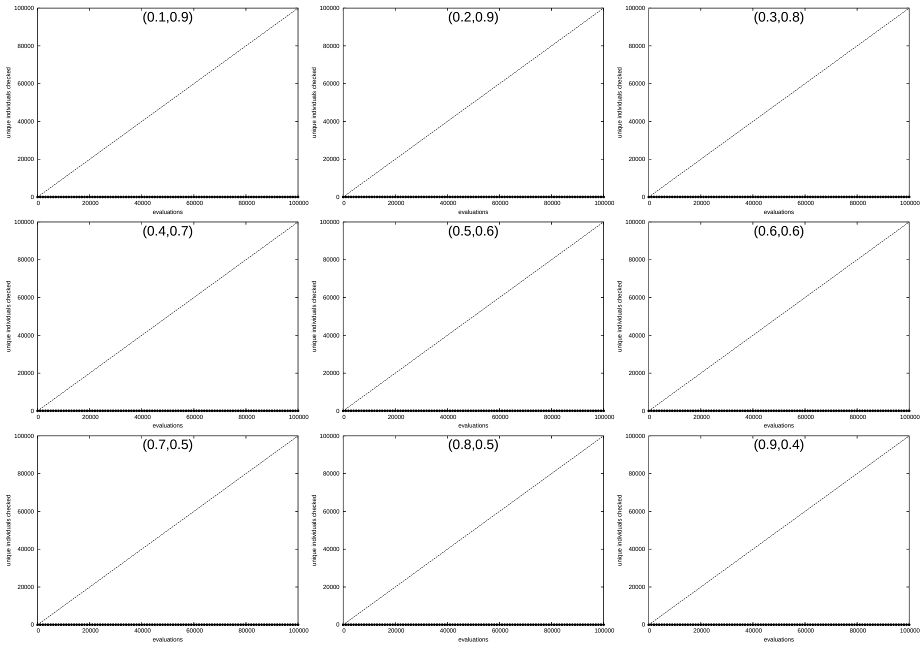

Figure 7.11: *UIC* of the *ArcEA3*.

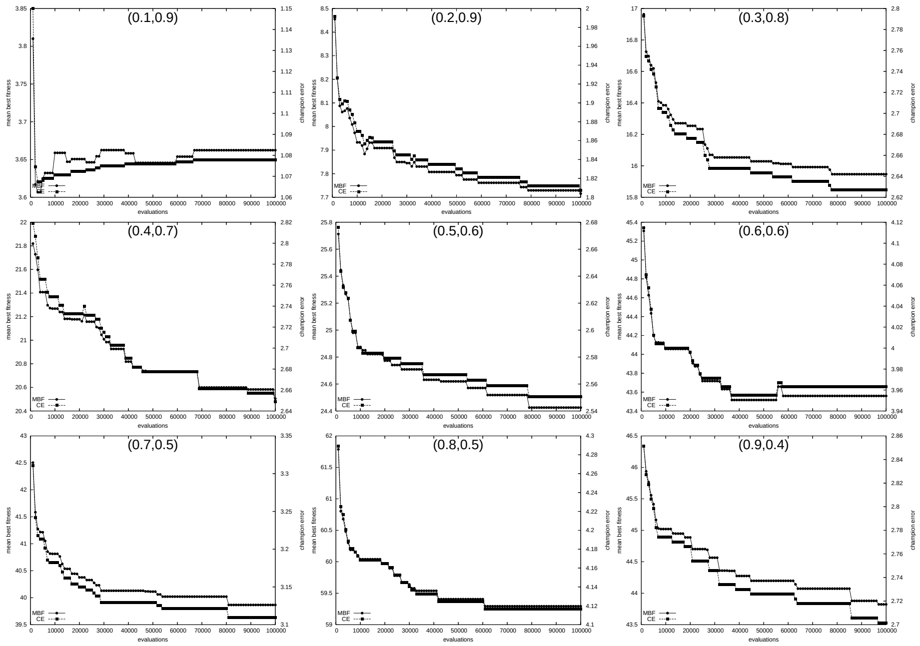

Figure 7.12: *MBF* and *MCE* of the *ArcEA3*.

7.3 *Co-evolutionary Algorithm*

The *Co-evolutionary Algorithm* (*CoeEA*) was proposed by J. Paredis, and was used to solve a number of problems: neural net learning ([71]), constraint satisfaction ([70, 71]), and searching for cellular automata that solve the density classification task ([72]). The *Co-evolutionary Algorithm* uses the co-evolutionary approach for evolutionary algorithms, from which it takes its name. The co-evolutionary approach pits two populations, commonly referred to as the host- and parasite-population, against each other.

The *Co-evolutionary Algorithm* for solving the constraint satisfaction problem uses a host-population of candidate solutions to compete with a parasite-population of constraints. All constraints of the CSP to be solved are included in the parasite-population. The size of the host-population is determined by a parameter. The fitness of an individual of both populations is based on a history of encounters between individuals of both populations. An *encounter* occurs when a constraint from the parasite-population is matched with the candidate solution of an individual of the host-population. If the constraint is satisfied in the candidate solution, the individual from the host-population earns a fitness point. If the constraint is not satisfied, the individual of the parasite-population earns a fitness point. The fitness value of an individual in both populations is the amount of fitness points gathered in the last 200 encounters. By matching often violated constraints with candidate solutions that have satisfied many constraints recently, a dynamic host-parasite relationship between the two populations is established. The relationship is characterised as a host-parasite relationship because both populations depend on each other for their fitness.

At each generation during the run of the *Co-evolutionary Algorithm*, 20 encounters between the individuals of the host- and parasite-population are allowed to occur. Encounters occur by repeatedly selecting an individual from each population and pairing them off. Selecting the individuals is biased forwards selecting individuals with higher fitness values. The genetic operators of crossover and mutation are applied only to the individuals of the host-population. The crossover operator is the two-point surrogate crossover operator, described in [87, 13]. The operator is designed to minimise the chance of generating children that have a similar candidate solution as their parents. The mutation operator used in the *Co-evolutionary Algorithm* is the uniform random mutation operator.

7.3.1 *CoeEA* Characteristics and Parameter Setup

Table 7.31 shows the characteristics table of the *Co-evolutionary Algorithm*. The *Co-evolutionary Algorithm* uses a steady state evolutionary model, an ordered set of values representation, a uniform random mutation operator, and a replace worst survivor selection operator, explained in Chapter 5. Selection of the individuals in both populations is done using the biased ranked parent selection operator. The fitness function and the two-point surrogate crossover operator used by the *Co-evolutionary Algorithm* have been discussed in the previous section.

Table 7.32 shows the parameters table for the *Co-evolutionary Algorithm*. The *Co-*

CoeEA	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	*CoeEA* Objective Function
Crossover operator	Two-point Surrogate Crossover
Mutation operator	Uniform Random Mutation
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	None

Table 7.31: Characteristics of the *CoeEA*.

CoeEA	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Individual History Size	200
Ranking Bias	1.5
Number of Encounters	20
Encounter Bias	1.5
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.32: Parameters of the *CoeEA*.

evolutionary Algorithm has a host-population of 10 individuals (Population Size), from which 10 parents are selected (Selection Size) using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias). The two-point surrogate crossover operator is applied with a crossover rate of 1.0 (Crossover Rate) and the uniform random mutation operator uses a mutation rate of 0.1 (Mutation Rate). Experiments with the *CoeEA* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations). Each individual in both populations maintains a history of 200 encounters (Individual History Size) and each generation of the *Co-evolutionary Algorithm*, 20 encounters are performed (Number of Encounters). Selection of the individuals from both populations for these encounters is done using the biased ranking parent selection operator, using a bias of 1.5 (Encounter Bias).

7.3.2 *CoeEA* Experimental Results

Table 7.33 shows that the *Co-evolutionary Algorithm* is unable to solve the CSP instances in the mushy region in any of its runs nor for a sizable portion of the solvable region. Consequently, the *AES* and *CC* for these density-tightness combinations are undefined in Tables 7.34 and 7.35. These tables also show that the *Co-evolutionary Algorithm* uses a lot of *AES* and *CC* when the run is successful. We believe that one reason for this performance is that the host-parasite relationship between the two populations is too dynamic, even with the long history of encounters used. This can result in the best individual in the host-population satisfying the constraint that has been violated recently the most in one generation but violating it in the next. This dynamic relationship of the two populations can result in constant changes to both populations without ever resulting in a directed search to a global optimum, an example of the *Red Queen*-principle [86]. In experiments not shown here, we tried to fine-tune the parameters of the *Co-evolutionary Algorithm*, in an effort to increase the performance of the algorithm. This was unsuccessful.

The *UIC* plots for the *Co-evolutionary Algorithm* in Figure 7.13 show that the algorithm searches through a large portion of the search space for the CSP instances in the mushy region. However, the *MBF/MCE* plots in Figure 7.14 show that for all the new unique individuals checked, no increase, on average, was achieved in the *MBF* or the *MCE*. In fact, the *UIC* and the *MBF/MCE* plots together suggest the behaviour of a random search algorithm. This means that the fitness values calculated by the encounters of the host- and parasite-population is of no use to maintain selection pressure. Although many unique individuals are checked, probably because of the use of the two-point surrogate crossover operator, the information gained by evaluating these individuals is not used to produce individuals with a higher fitness value in the next generation. Without selection pressure, the *CoeEA* can not direct the search to a global optimum, i.e., a solution.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	0.92	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	0.524	0.952	1.0	1.0	0.96	0.664	0.316	0.16	0.052
0.3	0.18	0.252	0.78	0.344	0.084	0.02	0.0	0.004	0.0
0.4	0.092	0.02	0.008	0.024	0.0	0.0	0.0	0.0	0.0
0.5	0.016	0.0	0.0	0.0	0.0	0.0	0.0	0.0	—
0.6	0.008	0.0	0.0	0.0	0.0	0.0	—	—	—
0.7	0.0	0.0	0.0	0.0	—	—	—	—	—
0.8	0.0	0.0	0.0	—	—	—	—	—	—
0.9	0.0	0.0	—	—	—	—	—	—	—

Table 7.33: *SR* of the *CoeEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	583	868	1220	1426	2007	2706	3817	6107	8534
0.2	499	3248	6024	13263	29972	42686	50073	54494	48403
0.3	266	3682	35468	51860	48209	34436	*undef.*	28010	*undef.*
0.4	10	292	34610	63367	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*
0.5	10	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	—
0.6	10	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	—	—	—
0.7	*undef.*	*undef.*	*undef.*	*undef.*	—	—	—	—	—
0.8	*undef.*	*undef.*	*undef.*	—	—	—	—	—	—
0.9	*undef.*	*undef.*	—	—	—	—	—	—	—

Table 7.34: *AES* of the *CoeEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	4059	9530	19502	28494	50156	78444	129743	232063	366927
0.2	3476	35708	96364	265242	749275	$1 \cdot 10^6$	$2 \cdot 10^6$	$2 \cdot 10^6$	$2 \cdot 10^6$
0.3	1842	40484	567463	$1 \cdot 10^6$	$1 \cdot 10^6$	998624	*undef.*	$1 \cdot 10^6$	*undef.*
0.4	50	3192	553740	$1 \cdot 10^6$	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*
0.5	50	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	—
0.6	50	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	—	—	—
0.7	*undef.*	*undef.*	*undef.*	*undef.*	—	—	—	—	—
0.8	*undef.*	*undef.*	*undef.*	—	—	—	—	—	—
0.9	*undef.*	*undef.*	—	—	—	—	—	—	—

Table 7.35: *CC* of the *CoeEA*.

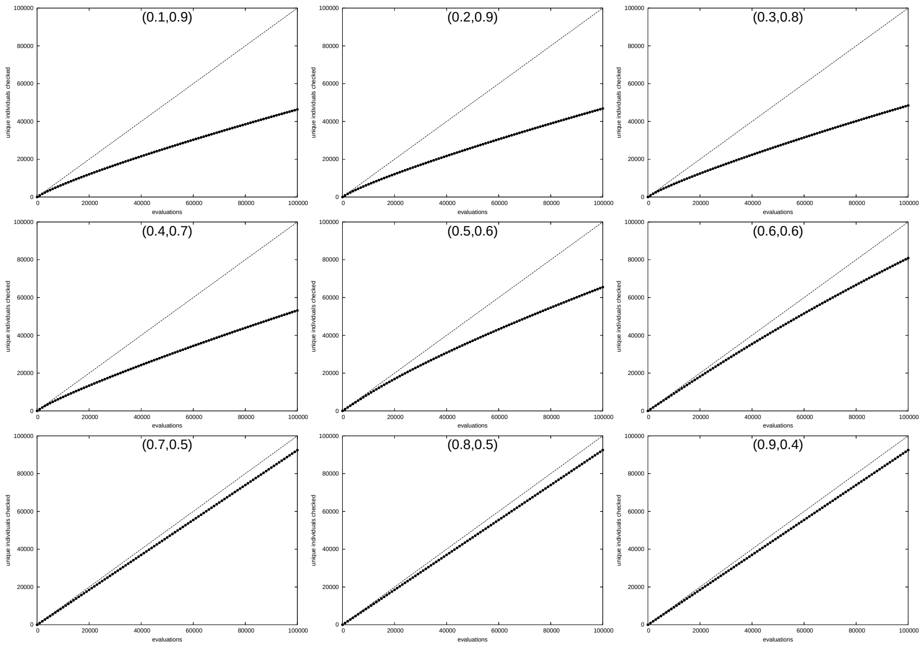

Figure 7.13: *UIC* of the *CoeEA*.

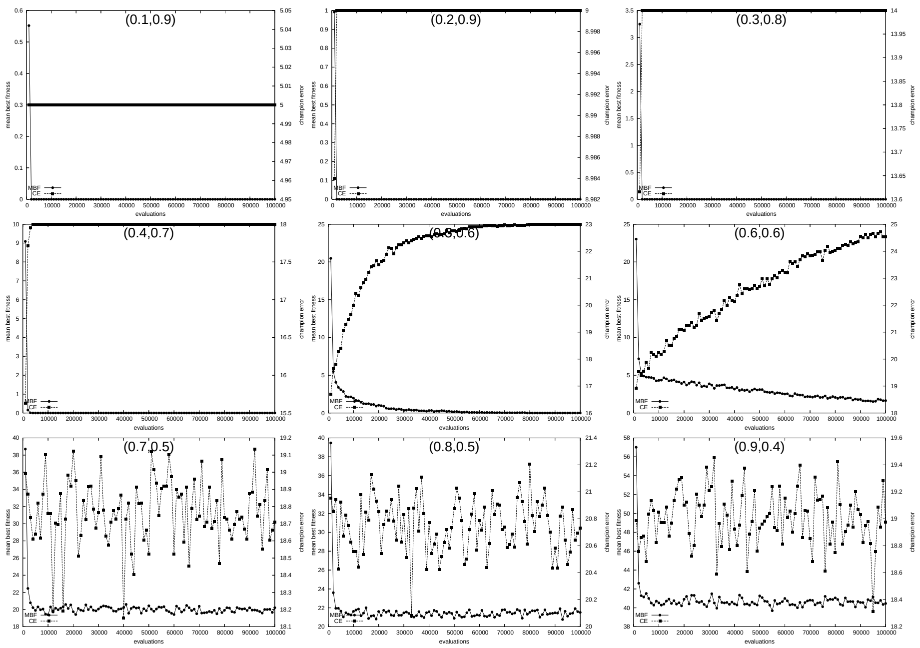

Figure 7.14: *MBF* and *MCE* of the *CoeEA*.

7.4 *Eliminate-Split-Propagate Evolutionary Algorithm*

In [57], E. Marchiori introduced an evolutionary algorithm for solving constraint satisfaction problems based on pre- and post-processing techniques for CSPs. The algorithm was further investigated in [19, 85], but here we use the version from [57]. We call this algorithm the *Eliminate-Split-Propagate Evolutionary Algorithm* (*ESPEA*). The technique applied in the *Eliminate-Split-Propagate Evolutionary Algorithm* is based on the glass-box approach from [85] which decomposes a CSP in such a way that there is only one single type of constraint. By decomposing more complex constraints into primitive ones, the resulting constraints have the same granularity and therefore the same intrinsic hardness. The rewriting of constraints is done in two steps and is called *constraint processing*. Because after constraint processing, all constraints have the same form, a single repair rule can be used to enforce *dependency propagation*. Because a single repair rule is used, a local-search technique can be used to repair an individual, applying the repair rule to every violated constraint in a candidate solution. The *ESPEA* takes its name from the initials of the two steps of the constraint processing phase, *Eliminate* and *Split*, and from the propagation of the dependencies by the repair rule. Constraint processing and dependency propagation is further discussed below:

Constraint Processing When the *ESPEA* was introduced it was tested on the *five houses puzzle* and the N-queens problem ([57, 85]). The constraints in these problems are, unlike the definition of constraints in the CSP, often defined as equations. These equations are the equivalent of what would be several constraints in the CSP. Because the equations combine several constraints, their relative complexity varies. Constraint processing is a way of reducing the variance of complexity of these constraints. The method proposed for processing these constraints consists of two phases: the *elimination phase* and the *split phase*. The elimination phase eliminates functional constraints in order to reduce the number of variables in the problem analogous to the GENOCOP method ([64]). The split phase then decomposes the resulting constraints into a set of constraints in canonical form. Each constraint is represented by a composition of primitive constraints. The canonical form proposed in [57] is of the form:

$$\alpha \cdot x_i - \beta \cdot x_j \neq \gamma \tag{7.1}$$

where x_i and x_j stand for the variables of the constraint. Because some variables are discarded during the elimination phase, when the solution of the original CSP is calculated, these variables have to be recovered. This in effect, reverses the elimination phase. Because we use a constraint satisfaction problem without functional variables and with constraints already in a canonical form, constraint processing is unnecessary, although the dependency propagation step has to be rewritten using these constraints.

Dependency Propagation Dependency propagation is implemented in the form of a probabilistic repair rule:

$$\textbf{if } \alpha \cdot p_i - \beta \cdot p_j = \gamma \textbf{ then } \text{re-label } p_i \text{ or } p_j \tag{7.2}$$

> The repair rule deals with violations of primitive constraints. It states that if a constraint is violated by a candidate solution, it should either re-label the first or the second variable of the constraint. There are three issues to resolve with this repair rule: which variable to re-label, to which value of the variable's domain to re-label it to, and in what order the constraints are to be processed. In [57], a uniform randomly chosen variable is re-labelled with a uniform randomly chosen value. The constraints are checked in random order. No bias is applied to any of these choices nor to the ordering of the constraints.

Because in our definition of the CSP, the constraint processing step of the *Eliminate-Split-Propagate Evolutionary Algorithm* is unnecessary, this leaves only the dependency propagation step. This is implemented as a repair rule. The repair rule is implemented in a repair operator added to the genetic operators of the *Intuitive Evolutionary Algorithm*. The repair operator is used after the mutation operator. The other components of the *Intuitive Evolutionary Algorithm* remain unchanged.

7.4.1 *ESPEA* Characteristics and Parameter Setup

Table 7.36 shows the characteristics table of the *Eliminate-Split-Propagate Evolutionary Algorithm*. The characteristics of the *Eliminate-Split-Propagate Evolutionary Algorithm* are for a large part identical to the characteristics of the *Intuitive Evolutionary Algorithm* in that it too uses a steady state evolutionary model, an ordered set of values representation, the f_1 fitness function, the uniform random crossover operator, the uniform random mutation operator, the bias ranking parent selection operator and the replace worst survivor selection operator. All these characteristics are explained in Chapter 5. As an additional operator, the *Eliminate-Split-Propagate Evolutionary Algorithm* uses the *ESPEA* repair operator discussed in the previous section.

Table 7.37 shows the parameter table of the *Eliminate-Split-Propagate Evolutionary Algorithm*. The *Eliminate-Split-Propagate Evolutionary Algorithm* has a population of 10 individuals (Population Size), from which 10 parents are selected (Selection Size) using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias). The crossover operator of the *Eliminate-Split-Propagate Evolutionary Algorithm* is applied with a crossover rate of 1.0 (Crossover Rate) and the uniform random mutation operator in the *Eliminate-Split-Propagate Evolutionary Algorithm* uses a mutation rate of 0.1 (Mutation Rate). The experiments of the *Eliminate-Split-Propagate Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations). The *ESPEA* repair operator has only a single parameter, determining the portion of constraints that are checked to repair the individuals. We use all constraints to repair the individuals: 1.0 (Constraints Check Rate).

ESPEA	
Evolutionary Model	Steady State
Representation	Ordered Set of Values
Objective Function	f_1
Crossover operator	Uniform Random Crossover
Mutation operator	Uniform Random Mutation
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	Repair Operator

Table 7.36: Characteristics of the *ESPEA*.

ESPEA	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Constraints Check Rate	1.0
Ranking Bias	1.5
Crossover Rate	1.0
Mutation Rate	0.1

Table 7.37: Parameters of the *ESPEA*.

7.4.2 *ESPEA* Experimental Results

Table 7.38 shows that the *Eliminate-Split-Propagate Evolutionary Algorithm* solves the CSP instances in the solvable region in almost all runs. Only for density-tightness combinations $(0.3, 0.7)$, $(0.6, 0.5)$, and $(0.8, 0.5)$ was the *ESPEA* *SR* not almost 1.0. The *SR* of the *ESPEA* was not so high in the mushy region, only for density-tightness combination (0.1,0.9) did it have a *SR* of 1.0. The lowest *SR* of the *ESPEA* in the mushy region is for density-tightness combination (0.7,0.5) with a *SR* of 0.328. Tables 7.39 and 7.40 show that for the solvable region, the *ESPEA* had a fairly low *AES* and *CC*, for the density-tightness combinations in the mushy region however, the *ESPEA* uses a fairly large amount of both *AES* and *CC*. However, because these measures are calculated over successful runs only, and the *ESPEA* has a lower *SR* in the mushy region, these values are inaccurate. In general, the repair operator of the *ESPEA*, even though it does not use any expensive heuristics, still uses a certain amount of *CC* because all constraints are used to repair the individuals in the population.

The *UIC* plots of the *ESPEA* in Figure 7.15 show that the *ESPEA* searches through a substantial portion of the search space. The jump in the *UIC* plot for density-tightness combination (0.1,0.9) is explained by the fact that in between the two intervals, many runs of the algorithm were successful. Since the *UIC* is calculated as an average over all runs, this has an effect of the *UIC* as a whole. The *UIC* plots show that the *ESPEA* shows no sign of premature convergence of the population in the mushy region, enough new unique individuals are evaluated during the run of the algorithm. The *MBF/MCE* plots of the *ESPEA* in Figure 7.16 show that the f_1 objective function is similar to the calculation of the *MCE*. The *MBF* and the *MCE*, on average, follow each other closely. The plots further show that, except for density-tightness combination (0.1,0.9), the search concentrates rapidly around individuals that have the same fitness value. The exception for density-tightness combination (0.1,0.9) is explained by the fact that all runs of the *ESPEA* were successful after only a few evaluations.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	0.996	0.844	0.432
0.5	1.0	1.0	1.0	1.0	0.988	0.788	0.328	0.468	—
0.6	1.0	1.0	1.0	0.968	0.436	0.404	—	—	—
0.7	1.0	1.0	0.796	0.436	—	—	—	—	—
0.8	1.0	0.932	0.388	—	—	—	—	—	—
0.9	1.0	0.676	—	—	—	—	—	—	—

Table 7.38: *SR* of the *ESPEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	12	14	16	17	19	19
0.2	10	12	16	19	22	24	30	35	47
0.3	12	17	22	27	39	55	88	126	213
0.4	14	21	34	55	99	523	1832	5598	8365
0.5	19	30	64	126	2094	5972	8599	5332	—
0.6	24	50	162	2265	7928	5581	—	—	—
0.7	35	123	2854	6280	—	—	—	—	—
0.8	166	1157	4982	—	—	—	—	—	—
0.9	997	6604	—	—	—	—	—	—	—

Table 7.39: *AES* of the *ESPEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	50	91	162	246	401	603	786	996	1164
0.2	52	132	308	491	773	1039	1603	2154	3414
0.3	65	220	467	805	1579	2683	5325	8709	17026
0.4	90	294	803	1794	4318	27996	116941	402717	685535
0.5	139	441	1642	4356	96098	322239	549994	383535	—
0.6	187	817	4392	81351	364466	301103	—	—	—
0.7	300	2129	79780	225890	—	—	—	—	—
0.8	1614	20743	139361	—	—	—	—	—	—
0.9	9918	118774	—	—	—	—	—	—	—

Table 7.40: *CC* of the *ESPEA*.

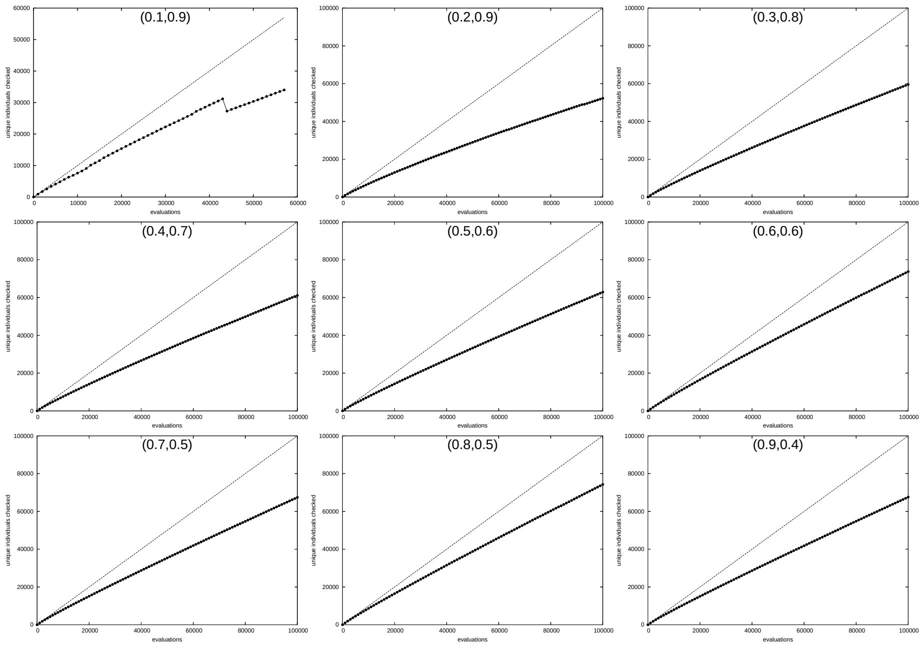

Figure 7.15: *UIC* of the *ESPEA*.

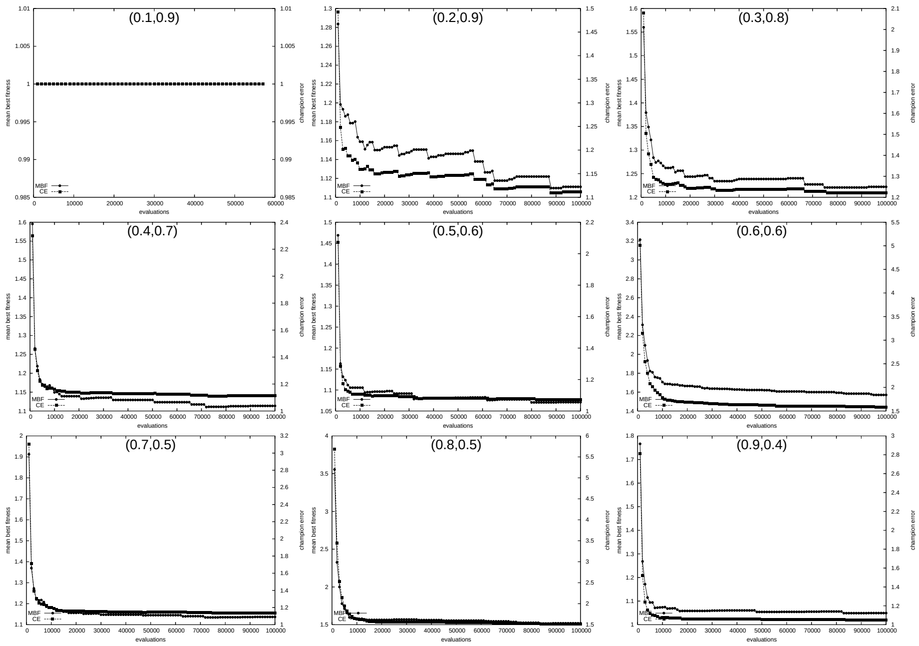

Figure 7.16: *MBF* and *MCE* of the *ESPEA*.

7.5 *Host-Parasite Evolutionary Algorithm*

In [45, 44], H. Handa *et al.* introduce an evolutionary algorithm also based on the co-evolutionary approach, which we call the *Host-Parasite Evolutionary Algorithm* (*HPEA*). In the *HPEA*, a parasite-population of schemata is used to improve a host-population of candidate solutions. Schemata are defined as candidate solutions where a number of variables are labelled with an asterisk. The asterisk is used as a "don't care"-value. The schemata are used as an overlay or template over the individuals of the host-population. When applied to a host-individual, the asterisk values in the schemata are replaced by the values of the corresponding variables of the host-individual.

Unlike the *Co-evolutionary Algorithm*, both host- and parasite-populations are evolved. Both populations have their own objective function and the evolution of both populations is done using genetic and selection operators. The schemata of the parasite-population are used to enhance the fitness of the host-population only. The relationship of the two populations is parasitic from the point of the parasite-population as the schemata and fitness values of the parasite-individuals depend solely on the host-individuals. However, it also resembles a symbiotic relationship as the parasite-population is used to enhance the ability of the host-population to find solutions to the problem. As such it resembles the relationship between, for example, sharks and their cleaner-fish.

The objective function of the host-population is based on the number of constraints violated by a candidate solution. The fitness of the host-individuals is normalised between zero and one and is to be maximised. The host-crossover operator is the uniform random crossover operator and the host-mutation operator is the uniform random mutation operator. Parents are selected using the biased ranking parent selection operator and survivors are selected using the replace worst survivor operator. With the exception of the different objective function, the host-part of the *HPEA* closely resembles the *Intuitive Evolutionary Algorithm*.

The fitness value of a parasite-individual is calculated by measuring the improvement of the schema on a portion of the host-population. The improvement is measured by summing the positive difference of the fitness values before and after the schema is applied to the host-individual. Applying a schema to a host-individual is called *super-positioning* the schema. The parasite-crossover operator is the uniform random crossover operator, the asterisk labels are treated like ordinary labels. The parasite-mutation operator is an adaptation of the uniform random mutation operator where for re-labelling to a asterisk a new parameter is used. The parameter determines the probability that an asterisk is used to re-label a variable, instead of an ordinary value. A third, repair, operator is added to evolve the parasite-population. The operator only considers the variables not labelled with an asterisk. These variables are re-labelled iteratively to values that do not violate any constraint relevant to other labelled variables. A local-search algorithm as was used in the *Hill Climber with Restart Algorithm* is used to do this. Parents for the parasite-population are selected using the biased ranking parent selection operator and survivors are selected using the replace worst survivor selection operator.

Interaction between the host population and the parasite population is based on two mechanisms:

Super-position Super-position is the interaction from the host-population to the parasite-population. This interaction provides the schemata in the parasite-population with their fitness values. Each schema in the parasite-population is applied to a number of host-individuals. Asterisk values in the schemata are replaced by the corresponding values of the host-individual.

Transcription Transcription is the interaction from the parasite-population to the host-population and is the actual transmission of the parasite-population's genetic information. The *Host-Parasite Evolutionary Algorithm* sequentially performs a generation of the host population before it performs a generation of the parasite population. Transcription is performed after the parasite population is evaluated. It uniform randomly selects a number of host-individuals based on a parameter called the transcription rate. Randomly selected schemata are then super-positioned over these host-individuals.

The *Host-Parasite Evolutionary Algorithm* uses two populations and in effect evolves these populations separately, only exchanging genetic information during super-position and transcription. Different genetic and selection operators and even objective functions can be used for the host part of the algorithm.

7.5.1 *HPEA* Characteristics and Parameter Setup

Table 7.41 shows the characteristics table for the *Host-Parasite Evolutionary Algorithm*. Unlike the other characteristics tables in this chapter, the table for the *Host-Parasite Evolutionary Algorithm* consists of three columns. The centre column show the characteristics of the host part of the algorithm and the right column shows the characteristics of the parasite part of the algorithm. Both the host and the parasite-population of the *HPEA* use a steady state evolutionary mode, a uniform random crossover operator, a uniform random mutation operator, a biased ranked parent selection operator and a replace worst survivor selection operator. The crossover operator and the mutation operator for the parasite-population have been adapted so that they can handle schemata, no adjustment is needed for the host-population's genetic operators since it uses an ordered set of values representation. These characteristics are explained in Chapter 5. As a third operator, the parasite part of the algorithm also includes a repair operator, described in the previous section. The host-population uses the f_1 objective function that normalises the fitness values to a range between 0 and 1, maximised. The objective function of the parasite-population is based on the improvement of the transcription of the schemata on a number of host-individuals, explained above. The term *Improvement* f_1 is used in the characteristics table to describe this. More details on these objective functions can be found in the previous section as well.

Table 7.42 shows the parameter table for the *Host-Parasite Evolutionary Algorithm*. The *Host-Parasite Evolutionary Algorithm* maintains a host-population of 10 individ-

HPEA		
Evolutionary Model	Steady State	Steady State
Representation	Ordered Set of Values	Schemata
Objective Function	f_1 Normalised	Improvement f_1
Crossover operator	Uniform Random Crossover	Uniform Random Crossover
Mutation operator	Uniform Random Mutation	Uniform Random Mutation
Parent Selection	Biased Ranking	Biased Ranking
Survivor Selection	Replace Worst	Replace Worst
Other Functions	None	Repair Operator

Table 7.41: Characteristics of the *HPEA*.

HPEA	
Host Population Size	10
Parasite Population Size	5
Host Selection Size	10
Parasite Selection Size	5
Maximum Number of Evaluations	$100,000$
Number of Super-Positions	2
Transcription Rate	0.8
Mutation Rate Host Population	0.1
Mutation Rate Parasite Population	0.3
Asterisk Rate	0.7
Ranking Bias Host	1.5
Ranking Bias Parasite	1.5
Crossover Rate Host Population	1.0
Crossover Rate Parasite Population	1.0

Table 7.42: Parameters of the *HPEA*.

uals (Host Population Size), from which 10 parents are selected (Host Selection Size) using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias Host). Simultaneously, the *Host-Parasite Evolutionary Algorithm* maintains a parasite population of 5 individuals (Parasite Population Size), from which 5 parents are selected (Parasite Selection Size) using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias Parasite). The crossover operators of both the host- and the parasite-population are applied with a crossover rate of 1.0 (Crossover Rate Host Population and Crossover Rate Parasite Population) and the mutation operator of both populations uses a mutation rate of 0.1 (Mutation Rate Host Population and Mutation Rate Parasite Population). The experiments of the *Host-Parasite Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations have been performed (Maximum Number of Evaluations), combining the number of evaluations of both the host- and the parasite-population. During each fitness evaluation of an individual of the parasite-population, it is super-positioned over 2 host population individuals (Number of Super-Positions). The *Host-Parasite Evolutionary Algorithm* uses a transcription rate of 0.8 (Transcription Rate) and the uniform random mutation operator for the parasite-population uses an asterisk rate of 0.7 (Asterisk Rate).

7.5.2 *HPEA* Experimental Results

Table 7.43 shows that the *HPEA* has reasonable *SR* for the solvable region of the test-set. In the mushy region however, the *SR* of the algorithm is much lower. Table 7.44 shows that the *AES* to attain this *SR* is quite large. As expected, maintaining the two populations of the *HPEA* uses many evaluations. Table 7.45 shows that the *HPEA* also needs a high *CC* to attain this *SR*. The low *SR* of the *HPEA* is also explained because of the lower number of allowed evaluations for the host part of the algorithm. Because the *HPEA* uses evaluations for the maintenance of both populations, and the runs are terminated after a certain number of evaluations have been used, the host-population of the algorithm is allowed fewer evaluations to find a solution in than the population of an algorithm with has only one population. This is a drawback of all evolutionary algorithms that use the co-evolutionary approach: the extra cost incurred by having to maintain two populations has to be compensated by an improved performance of the algorithm. The high *CC* of the *HPEA* is probably caused by the local-search technique used in the repair operator of the parasite-population. The *SR* of the *HPEA* is not increased enough to compensate for the high *CC* cost of this operator however.

The *UIC* plots in Figure 7.17 show that the *HPEA* searches only through a small portion of the search space. The amount of search space searched is probably limited by the way the parasite-population is used. The *MBF/MCE* plots in Figure 7.18 show that the *MBF* and *MCE* graphs follow each other closely. Except for density-tightness combination (0.1,0.9), the *SR* of the *HPEA* is low, which makes both the *MBF/MCE* and *MCE* measures accurate and explains the smooth monotonic decrease of both plots. Both plots together show that the population of the *HPEA* does not converge prematurely to a local optimum. The erratic behaviour of the *MBF/MCE* plot for density-tightness combination (0.1,0.9) is explained by the effects of successful runs on calculating the mean of the *MBF* and *MCE* measures.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.988	0.912
0.4	1.0	1.0	1.0	1.0	0.984	0.968	0.788	0.472	0.156
0.5	1.0	1.0	1.0	0.988	0.768	0.436	0.152	0.204	—
0.6	1.0	1.0	0.96	0.708	0.188	0.188	—	—	—
0.7	1.0	1.0	0.576	0.228	—	—	—	—	—
0.8	1.0	0.852	0.256	—	—	—	—	—	—
0.9	0.98	0.564	—	—	—	—	—	—	—

Table 7.43: *SR* of the *HPEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	25	25	26	28	33	43	51	69	75
0.2	25	29	44	72	103	139	189	242	380
0.3	27	50	98	164	253	394	851	2713	6514
0.4	35	82	199	339	1143	6881	10771	16288	20945
0.5	57	148	673	3512	11357	21170	21258	20629	—
0.6	86	263	3603	14406	20224	22063	—	—	—
0.7	143	2255	12752	20118	—	—	—	—	—
0.8	875	8692	23212	—	—	—	—	—	—
0.9	2727	15222	—	—	—	—	—	—	—

Table 7.44: *AES* of the *HPEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	100	180	316	478	788	1438	2001	3165	3836
0.2	100	307	1047	2268	4159	6194	9631	13298	22539
0.3	168	995	3191	6381	11352	19695	45607	151718	416938
0.4	413	2129	7267	14424	56944	367844	645799	$1 \cdot 10^6$	$1 \cdot 10^6$
0.5	1108	4453	25003	167218	630930	$1 \cdot 10^6$	$2 \cdot 10^6$	$2 \cdot 10^6$	—
0.6	2099	9045	156620	696238	$1 \cdot 10^6$	$1 \cdot 10^6$	—	—	—
0.7	4022	77272	593063	$1 \cdot 10^6$	—	—	—	—	—
0.8	30985	361459	$1 \cdot 10^6$	—	—	—	—	—	—
0.9	85374	718336	—	—	—	—	—	—	—

Table 7.45: *CC* of the *HPEA*.

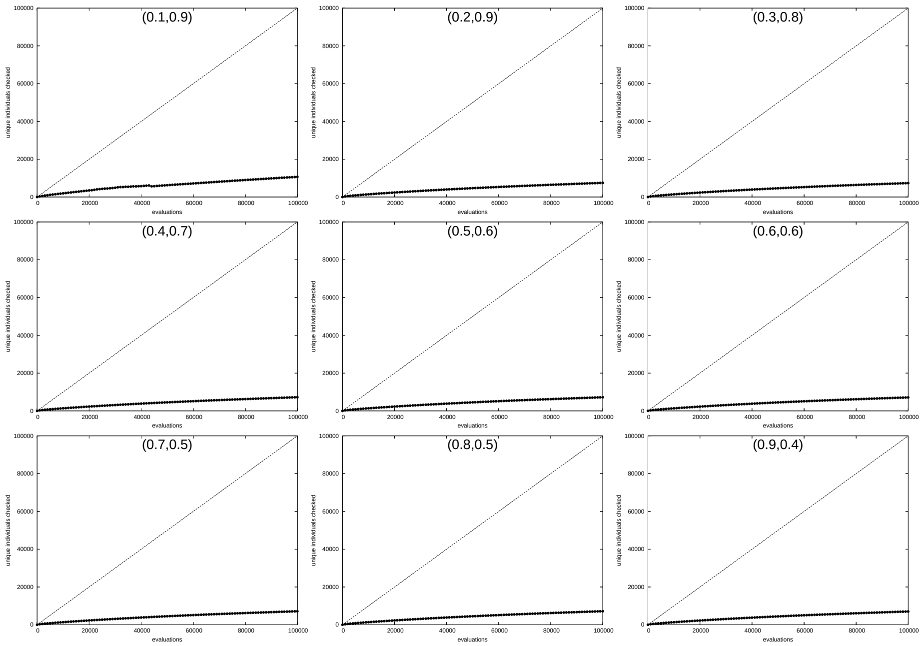

Figure 7.17: *UIC* of the *HPEA*.

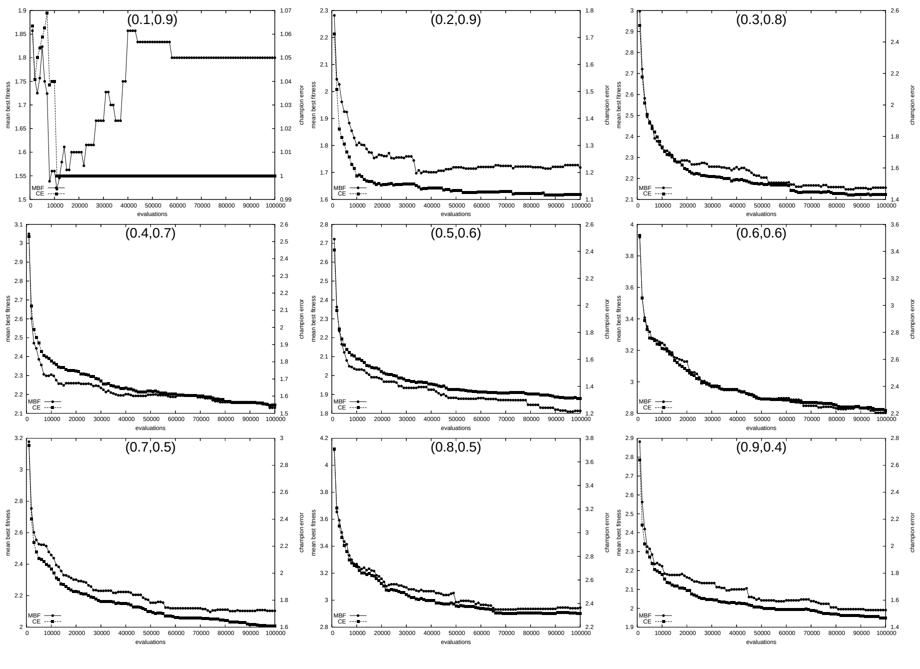

Figure 7.18: *MBF* and *MCE* of the *HPEA*.

7.6 *Local Search Evolutionary Algorithm*

In [59], E. Marchiori introduced another evolutionary algorithm that uses the combination of a repair operator and ordinary variation operator. The repair method consists of a specially adapted local-search algorithm. We call this algorithm: the *Local Search Evolutionary Algorithm* (*LSEA*). In [58], the algorithm was adapted to solve the Maximum Clique Problem, closely related to the CSP, and a comparison was given between an evolutionary algorithm setup, an iterated local-search setup, and a local-search setup with a restart strategy.

The *Local Search Evolutionary Algorithm* uses an array of domain sets representation. One domain set for each variable of the CSP is used. The idea is that the algorithm will reduce the domain sets to include only values that do not violate relevant constraints to the values in the other domain sets. During the search, more and more values are removed from the domain sets until only the values remain that are consistent with each other. Because only values remain in the domain sets that are consistent with the values in the other domain sets, the objective function of the *LSEA* is straightforward, it counts the *non*-empty domain sets in the individual. Since the *Local Search Evolutionary Algorithm* searches for individuals with domain sets with at least on value consistent with each other, this is enough. The objective function is called the LS objective function.

Because the representation used by the *Local Search Evolutionary Algorithm* is so different from the ordered set of values representation, the standard genetic operators cannot be used. New genetic operators were therefore designed. The *Local Search Evolutionary Algorithm* has three operators: the LS crossover operator, the LS mutation operator and the LS repair operator. The LS crossover operator takes two parents and generates two children. Initially the domain sets of the children are empty. With equal probability, each value from the domain sets of the parents is added to the corresponding domain set of either the first or the second child. No values are added twice to a domain set, instead, the value is added to the domain set of the child that does not contain it yet.

The LS mutation operator has two parts, it takes one parent to produce one child. The first part adds a uniform randomly chosen value to a uniform randomly chosen domain set of the child. If the value is already in the domain set, another value is chosen. The second part of the operator removes a value of a domain set. The value is selected with a low probability, typically 0.05. Neither the LS crossover operator nor the LS mutation operator uses heuristics and both operators are blind to constraints. The biased ranking selection operator is used for parent selection and replace worst survivor selection is used for survivor selection.

The LS repair operator is applied just after initialisation of the individuals and just after the mutation operator. It consists of three parts, called *initialisation*, *repair*, and *improve*. The local-search repair operator takes a single parent to construct a single child. The objective of the repair operator is to have the child contain a maximal partial solution over all variables of the CSP, constructed based on the parent. The three parts of the local-search repair operator are described below:

Initialisation The initialisation part of the operator initialises the child with empty domain sets for all variables of the CSP.

Repair The repair part of the operator consists of two phases:

- **Extract** During the extract phase the operator selects for each variable in the CSP a uniform randomly chosen value from the domain set of the parent. It then checks if this value is consistent with the other values already added to the child. If it is not consistent, another value is uniform randomly selected. No value can be selected twice. If no value is found to be consistent, the domain set is left empty. All domain sets are checked in random order.
- **Extend** During the extend phase, the operator tries to extend the domain sets of the child by checking if a uniform randomly chosen value *not* in the domain set of the parent is consistent with the already added values in the child. Again the different domain sets are extended in random order and no value in the domain sets is checked twice.

The objective of the repair part of the operator is to uniform randomly construct an array of maximal domain sets whose values are all consistent with each other.

Improve The improve part of the operator consists of three phases:

- **Arc-consistency** During the arc-consistency phase, the operator checks if there is a value in the domain sets that is inconsistent with all values of a (empty) domain set in the child. If such a value is in the domain sets of the child, it is removed. This phase is called arc-consistency because consistency is checked by arc.
- **Delete** During the delete phase, the operator removes the value in all domain sets that has the largest number of violated constraints relevant to the other domain set values. If two or more values have an equal number of violated constraints, all values are deleted.
- **Extend** This extend phase is the same as the extend phase in the repair part of the operator.

The objective of the improve part of the operator is to improve the array of domain sets by first eliminating values from the domain sets that cause one or more domain sets to remain empty and remove the values from the domain sets which limit the further extension of the child the most. After the arc-consistency and delete phase, the child is no longer an array of consistent maximal domain sets. The extend step is repeated in the hope that more values are added to the domain sets.

The domain sets and the values to be added to them are selected uniform randomly. This ensures that the array of consistent maximal domain sets is generated without bias. The operator also ensures that the algorithm remains in feasible search space, unlike the repair operator of the *Eliminate-Split-Propagate Evolutionary Algorithm*.

LSEA	
Evolutionary Model	Steady State
Representation	Array of Domain Sets
Objective Function	*LSEA* Objective Function
Crossover operator	LS crossover
Mutation operator	LS mutation
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	LS Repair Operator

Table 7.46: Characteristics of the *LSEA*.

LSEA	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Domain Value Add Rate	0.1
Domain Value Remove Rate	0.05
Repair Delete Rate	0.9
Ranking Bias	1.5
Crossover Rate	1.0

Table 7.47: Parameters of the *LSEA*.

7.6.1 *LSEA* Characteristics and Parameter Setup

Table 7.46 shows the characteristics table of the *Local Search Evolutionary Algorithm*. The *Local Search Evolutionary Algorithm* uses a steady state evolutionary model, the biased ranking parent selection operator and the replace worst survivor selection operator, explained in Chapter 5. The *Local Search Evolutionary Algorithm* uses the LS fitness function, the LS crossover operator, the LS mutation operator and the LS repair operator explained in the previous section.

Table 7.47 shows the parameters table of the *Local Search Evolutionary Algorithm*. The *Local Search Evolutionary Algorithm* uses a population of 10 individuals (Population Size), from which 10 parents are selected (Selection Size) using the biased ranked parent selection operator with a bias of 1.5 (Ranking Bias). The LS mutation operator adds a value to a domain set with a probability of 0.1 (Domain Value Add Rate) and removes a value from a domain set with a probability of 0.05 (Domain Value Remove Rate). The LS crossover operator is applied with a crossover rate of 1.0 (Crossover Rate). The LS repair operator deletes values from the domain sets with a probability of 0.9 (Repair Delete Rate). The experiments of the *Local Search Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations).

7.6.2 *LSEA* Experimental Results

Table 7.48 shows that the *LSEA* will find a solution for the CSP instances in the solvable region in almost every run. In the mushy region, the *SR* was lower but still comparatively high. Table 7.49 shows that the *AES* of the *LSEA* in the mushy region is low, finding on average a solution in the first generation for most CSP instances in the mushy region. The *AES* used for solving the CSP instances in the mushy region is higher but is comparatively low when compared to the other algorithms discussed. Table 7.50 shows that although the *LSEA* uses few *AES*, it uses many *CC*s. This indicates that most of the conflict checks are used outside the objective function. Since the other operators of the algorithm do not use conflict checks, these must all be used by the LS repair operator.

The *UIC* plots in Figure 7.19 show that the *LSEA* searches only a small portion of the search space. This is probably caused by the LS repair operator which ensures that the search is limited to the feasible search space only. The *MBF/MCE* plots in 7.20 show almost no difference between the *MBF* and *MCE* measures during the run. For density-tightness combination (0.1,0.9), all runs were successful before the first interval, so these plots show only a single data point. The flatness of the *MBF/MCE* plots is caused by the low number of *AES* needed by the *LSEA* to find a solution. The way in which the fitness function is calculated results in a rather static *MBF* measure indicating a low selection pressure with little difference between good and bad individuals.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.936
0.5	1.0	1.0	1.0	1.0	1.0	1.0	0.776	0.796	—
0.6	1.0	1.0	1.0	1.0	0.924	0.752	—	—	—
0.7	1.0	1.0	0.992	0.808	—	—	—	—	—
0.8	1.0	1.0	0.812	—	—	—	—	—	—
0.9	1.0	0.988	—	—	—	—	—	—	—

Table 7.48: *SR* of the *LSEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	10	10	10	10	10	10	10
0.2	10	10	10	10	10	10	10	10	10
0.3	10	10	10	10	10	10	10	10	13
0.4	10	10	10	10	10	13	24	363	4097
0.5	10	10	10	11	25	389	11562	11422	—
0.6	10	10	13	88	10124	12080	—	—	—
0.7	10	11	1399	5935	—	—	—	—	—
0.8	10	26	9825	—	—	—	—	—	—
0.9	13	540	—	—	—	—	—	—	—

Table 7.49: *AES* of the *LSEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	840	868	921	951	1003	1047	1108	1158	1231
0.2	895	974	1146	1253	1436	1656	1961	2307	2791
0.3	974	1167	1477	2013	2560	3174	4066	4912	6813
0.4	1103	1602	2440	3220	4551	6468	13073	164325	$2 \cdot 10^6$
0.5	1367	2306	3794	5347	13398	163912	$5 \cdot 10^6$	$4 \cdot 10^6$	—
0.6	1835	3390	6752	40279	$4 \cdot 10^6$	$5 \cdot 10^6$	—	—	—
0.7	2878	5545	619586	$3 \cdot 10^6$	—	—	—	—	—
0.8	4539	14481	$5 \cdot 10^6$	—	—	—	—	—	—
0.9	8893	300212	—	—	—	—	—	—	—

Table 7.50: *CC* of the *LSEA*.

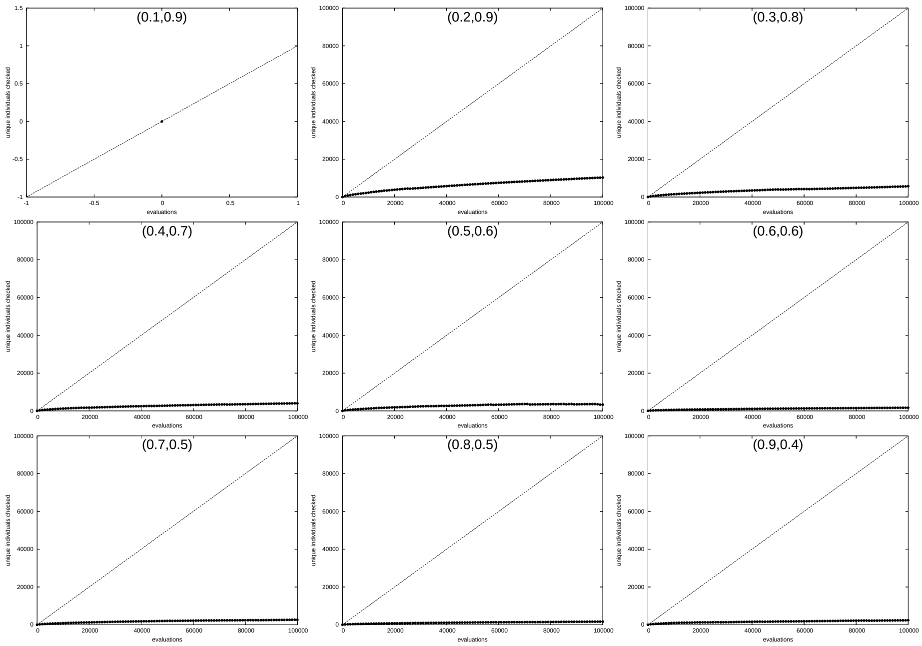

Figure 7.19: *UIC* of the *LSEA*.

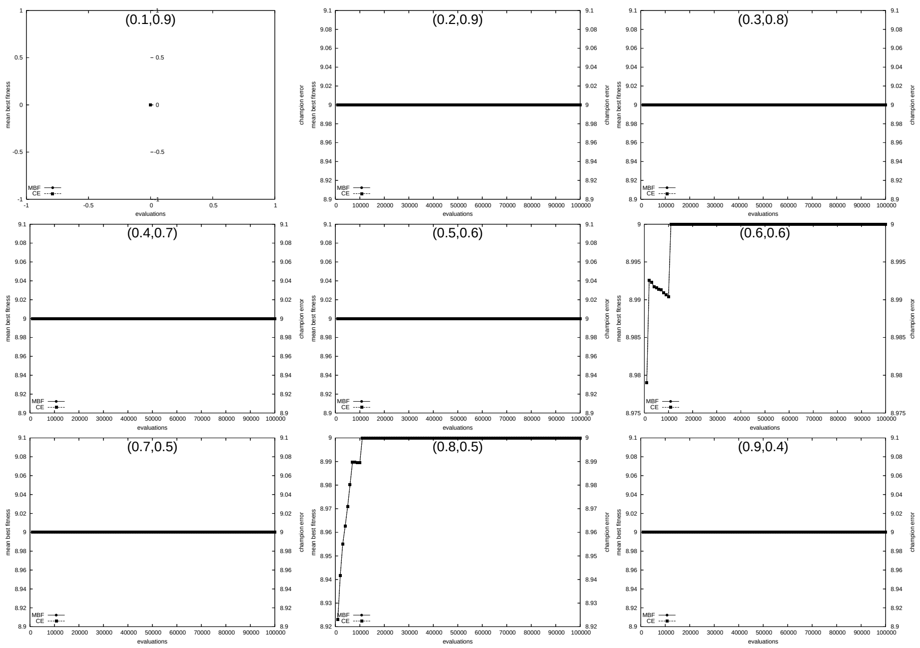

Figure 7.20: *MBF* and *MCE* of the *LSEA*.

7.7 *Micro-genetic Iterative Descent Evolutionary Algorithm*

The *Micro-genetic Iterative Descent Evolutionary Algorithm* (*MIDEA*) was proposed by G. Dozier *et al.* in [24] and was further refined in [14, 25]. In the *MIDEA*, information about the CSP is incorporated in both genetic operators and in the objective function. The objective function is adaptive and employs the Breakout Creating Mechanism developed by Morris in [66] to escape from local optima. The *Micro-genetic Iterative Descent Evolutionary Algorithm* is called micro-genetic because small populations are evolved.

The *MIDEA* uses a representation that includes a pivot value, the number of constraint violations for each variable, and a h-value additional to the ordered set of values representation. The h-value is used to determine the pivot variable of the individual. The pivot variable is initialised to zero.

The population is evolved using one of two genetic operators. Which operator is used is determined by an adaptive scheme. At initialisation of the algorithm, both operators have an equal probability of being used. After the operator is applied, the fitness values of the children are compared to the fitness values of the parents. If the child fitness values are better than the fitness values of the parents, the probability of using the operator is increased proportionally to the amount of the improvement. Each genetic operator has its own probability, called the *accumulated awards* of the operator. The probability of using the operator is calculated by dividing the accumulated award by the total accumulated awards of both operators.

The *MIDEA* uses the *multiple-point heuristic* operator ([26]) as a crossover operator. The operator recombines two parents into one child. The operator copies every value from the parent which are consistent with each other. The remaining variables are added by performing a multi-point crossover with probability $0.5 \cdot (1 + 1/\text{constraint violations}(\text{value}))$, or by copying the value from the first parent. The multi-point crossover chooses a value from a domain limited by the values of the two parents. As the domains of the variables are discrete, all values between the values of the parents can be selected. For a variable with first parent value 9 and second parent value 3, the operator can choose a value from the set $\{3, 4, 5, 6, 7, 8, 9\}$.

The *MIDEA* uses the *single-point heuristic mutation* operator. The operator re-labels a single variable. Which variable is re-labelled is determined by the pivot value of the parent. The variable is re-labelled to a value chosen uniform randomly from the family-domain of the variable, described below. The child is then compared to its parent. If the fitness value of the parent is better or equal to the fitness value of the child, the h-value of the pivot variable of the child is decreased by one and the child is inspected to see if the pivot should point to another variable. This is done by calculating the s-value of each variable. The s-value of variable is the sum of the number of constraint violations of the variable and its h-value. The variable with the highest s-value will be the new pivot variable of the child. If the current pivot variable has an equal s-value to one or more other variables, the pivot remains unchanged. If the s-values of other variables are equal, the pivot is chosen uniform randomly among them. If the fitness value of the child is better than the fitness value of the parent, the h-value and thus the pivot variable remains unchanged.

This method for inheriting information for choosing which variable is to be mutated provides two mechanisms for the algorithm to exploit. First, a consecutive line of successful children can optimise the number of constraint violations of a single variable. Second, it allows the algorithm to switch to other variables when this optimising stops or when other variables have higher s-values. A drawback of the method is that after a while it is possible that the h-values cause the algorithm to choose a variable that is not involved in any constraint violations. This occurs when the h-values of the variables involved in constraint violations get lower than the actual number of constraint violations. When this happen, no further progress will be made, and to prevent this, all h-values will be reset to zero using probability function r_i for individual i:

$$r_i = \frac{1}{|O_i| + 2} \tag{7.3}$$

where O_i is the number of variables involved in constraint violations caused by individual i.

The fitness value of an individual is determined by adding a penalty to the number of constraint violations of the individual. The penalty is the sum of the weights of all breakouts whose values occur in the individual. A breakout consists of two parts: a compound label that violates a constraint and a weight associated to the compound label. The set of breakouts is initially empty and is modified by increasing the weights of the breakouts or by adding new breakouts according to the technique used in the Iterative Descent Method ([66]).

In addition, the *Micro-genetic Iterative Descent Evolutionary Algorithm* uses the mechanism of maintaining families. The algorithm uses families to force the mutation operator into a more structured exploration of the search space. Each individual evaluated by the algorithm is assigned to a family. Each family has a domain for the pivot variables from which the mutation operator may choose when the pivot variable is re-labelled. Initially, a family starts this domain equal to the domain of the corresponding variable. When a value is used to label a family member, that value is removed from the domain set. This prevents future relative to reuse it. When a domain becomes empty, a new pivot variable is chosen and a new family is founded, having a full domain. The individual with the empty family domain becomes the first member of the new family.

7.7.1 *MIDEA* Characteristics and Parameter Setup

Table 7.51 shows the characteristics table of the *Micro-genetic Iterative Descent Evolutionary Algorithm*. The *Micro-genetic Iterative Descent Evolutionary Algorithm* uses a steady state evolutionary model, a biased ranking parent selection operator, and a replace worst survivor selection operator, explained in Chapter 5. The *MIDEA* uses a special *MIDEA* representation which adds breakouts to the f_1 objective function. The

MIDEA	
Evolutionary Model	Steady State
Representation	Special *MIDEA* Representation
Objective Function	$f_1$and Breakouts
Crossover operator	Multi-Point Heuristic
Mutation operator	Single-Point Heuristic
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	Families

Table 7.51: Characteristics of the *MIDEA*.

MIDEA	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Crossover Award	1
Mutation Award	1
Ranking Bias	1.5

Table 7.52: Parameters of the *MIDEA*.

MIDEA uses the multi-point heuristic operator as a crossover operator and the single-point heuristic operator as a mutation operator. The objective function and both genetic operators are explained in the previous section.

Table 7.52 shows the parameter table of the *Micro-genetic Iterative Descent Evolutionary Algorithm*. The *Micro-genetic Iterative Descent Evolutionary Algorithm* has a population of 10 individuals (Population Size), from which 10 parents are selected (Selection Size) using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias). The crossover operator and the mutation operator are applied based on an award system which awards one point for an application of the crossover operator when it improves the fitness of the individuals (Crossover Award) and one point for an application of the mutation operator when it improves the fitness of the individuals (Mutation Award). The experiments of the *Micro-genetic Iterative Descent Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations).

7.7.2 *MIDEA* Experimental Results

Table 7.53 shows that the *SR* of the *Micro-genetic Iterative Descent Evolutionary Algorithm* is low in both the solvable and the mushy region of the test-set. For the mushy region, the *MIDEA* did not find a solution in any run for five density-tightness combinations. Table 7.54 and 7.55 therefore show undefined entries for these density-tightness combinations. Given that the *SR* of the *MIDEA* is so low, both the *AES* and *CC* are inaccurate since their average is calculated only over a few successful runs. Still, both tables show that the *MIDEA* uses a large *AES* and *CC* to find solutions to the CSP instances in the test-set.

The *UIC* plots in Figure 7.21 show that the *MIDEA* searches through a small portion of the search space and that the *UIC* hardly increases during the run. This suggests premature convergence of the population on a local optimum. The *MBF/MCE* plots in Figure 7.22 support this suggestion as the plots show almost no variation in both the *MBF* and the *MCE*. Both the *UIC* and the *CC* plots are accurate because of the large number of unsuccessful runs. Combining the two plots we must conclude that, on average, the population of the *MIDEA* converges to a local optimum almost immediately after it is started.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.996
0.2	1.0	1.0	1.0	1.0	1.0	0.976	0.956	0.884	0.772
0.3	1.0	1.0	0.976	0.944	0.796	0.692	0.548	0.332	0.14
0.4	1.0	0.996	0.896	0.732	0.36	0.14	0.044	0.024	0.0
0.5	0.996	0.928	0.532	0.284	0.06	0.02	0.0	0.0	—
0.6	0.996	0.672	0.16	0.036	0.0	0.004	—	—	—
0.7	0.888	0.24	0.012	0.004	—	—	—	—	—
0.8	0.544	0.052	0.0	—	—	—	—	—	—
0.9	0.22	0.004	—	—	—	—	—	—	—

Table 7.53: *SR* of the *MIDEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	11	13	17	26	32	42	50
0.2	10	14	29	47	77	105	189	244	349
0.3	13	33	70	137	240	394	601	613	978
0.4	19	69	201	259	1305	641	4739	575	*undef.*
0.5	38	138	331	1601	1655	1200	*undef.*	*undef.*	—
0.6	72	221	502	661	*undef.*	6635	—	—	—
0.7	185	785	803	940	—	—	—	—	—
0.8	354	375	*undef.*	—	—	—	—	—	—
0.9	413	550	—	—	—	—	—	—	—

Table 7.54: *AES* of the *MIDEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1350	1361	1501	1777	2246	3499	4255	5638	6778
0.2	1404	1928	3942	6297	10338	14239	25465	32979	47118
0.3	1766	4434	9506	18490	32405	53283	81215	82753	132108
0.4	2576	9250	27192	34982	176355	86643	640305	77633	*undef.*
0.5	5086	18687	44748	216372	223653	162108	*undef.*	*undef.*	—
0.6	9670	29810	67776	89295	*undef.*	896603	—	—	—
0.7	24958	106109	108495	126945	—	—	—	—	—
0.8	47830	50687	*undef.*	—	—	—	—	—	—
0.9	55753	74250	—	—	—	—	—	—	—

Table 7.55: *CC* of the *MIDEA*.

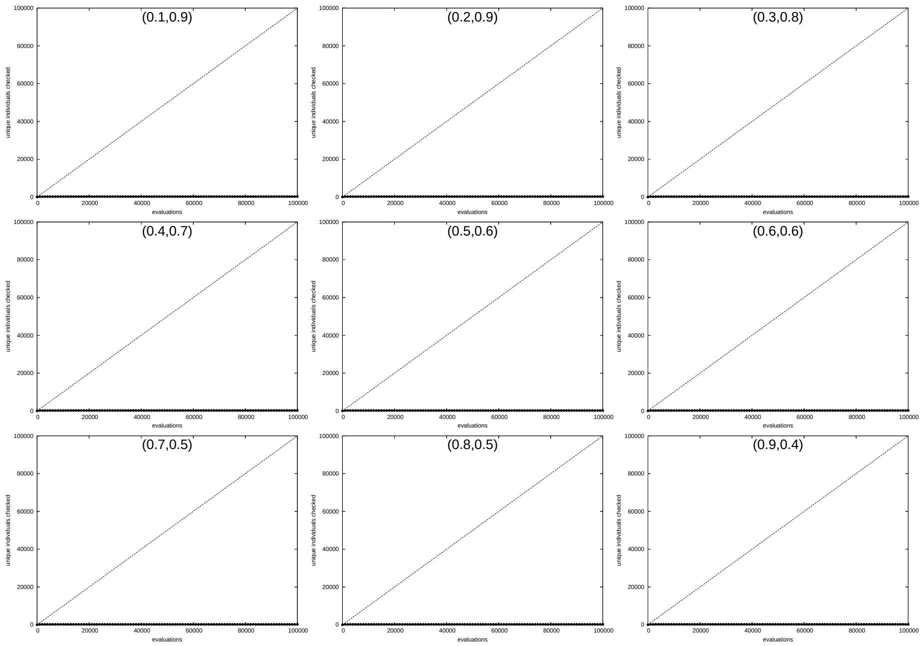

Figure 7.21: *UIC* of the *MIDEA*.

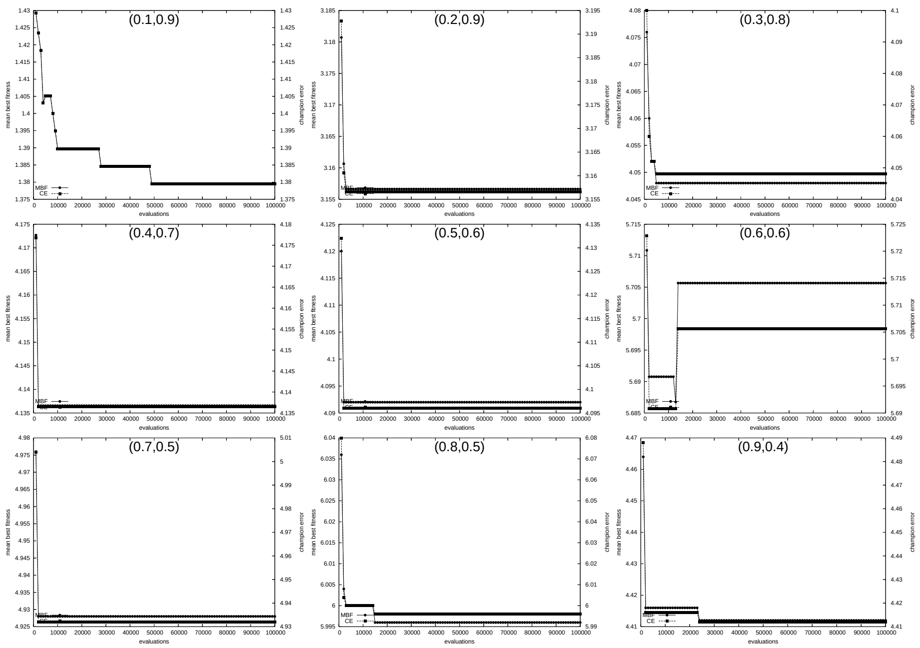

Figure 7.22: *MBF* and *MCE* of the *MIDEA*.

7.8 *Stepwise Adaptation of Weights Evolutionary Algorithm*

The *Stepwise Adaptation of Weights Evolutionary Algorithm* (*SAWEA*) was first introduced by A.E. Eiben and J.K. van der Hauw in [33, 84] as improvement to the weight adaptation mechanism of Eiben, Raué, and Ruttkay, defined in [30, 31]. The *Stepwise Adaptation of Weights Evolutionary Algorithm* has been studied in several variations in [30, 34, 35], and a comprehensive study of different parameters and genetic operators can be found in [17]. In [42], the *Stepwise Adaptation of Weights Evolutionary Algorithm* is surpassed by other techniques for specific suites of satisfiability problems (SAT), but for the constraint satisfaction problem, the *Stepwise Adaptation of Weights Evolutionary Algorithm* has been found to have good performance for different constraint satisfaction problems.

The *Stepwise Adaptation of Weights Evolutionary Algorithm* defines two equally important additions to the standard evolutionary algorithm: the decoder, and the stepwise adaptation of weights mechanism.

The decoder in the *Stepwise Adaptation of Weights Evolutionary Algorithm* takes a permutation of the variables of a constraint satisfaction problem and uses a greedy algorithm to label these variables, in order, with values from the domains of these variables, so that the thus constructed partial candidate solution remains consistent. Variables that can not be labelled with a consistent value are left unlabelled. The fitness value of an individual is the number of variables that are left unlabelled.

The stepwise adaptation of weights mechanism is based on the notion that some constraints in the constraint satisfaction problem are harder to satisfy than others. Performance of an evolutionary algorithm can be improved by focussing on satisfying these constraints. It is assumed that constraints that have not been satisfied after a number of iterations of the evolutionary algorithm are hard to satisfy. The stepwise adaptation of weights mechanism uses this assumption by defining a special objective function: the *SAW objective function*.

The SAW objective function maintains a set of weights for each constraint in the constraint satisfaction problem. This set is initialised by a assigning a weight of 1 to each constraint. After an interval of a number of generations, the individual with the best fitness value in the population is used to increase the weights of the constraints that are violated in the individual. Because the decoder labels only variables that are consistent with each other, constraints with an relevant unassigned variable are considered to be violated. The amount with which the weight is increased is determined by parameter Δw. Usually a value of $\Delta w = 1$ is used. The interval after which the weights are updated is determined by another parameter: the *update interval*. A commonly used value for the update interval parameter is 25 generations of the *SAWEA*.

In [17] for the constraint satisfaction problem, and in [34] for the k-graph colouring problem, it was found that there was no significant difference in the performance of the *Stepwise Adaptation of Weights Evolutionary Algorithm* when the fitness of an individual was calculated based on variables that were left unassigned instead of constraints that were violated. As such, we use the variable-weights variant of the *Stepwise Adaptation of Weights Evolutionary Algorithm* here. This means that the SAW objective function maintains a set of weights over all variables of the constraint satisfaction problem. The weights are increased when a variable is left unassigned by the decoder. The fitness value of an individual is calculated by adding the weights of all unassigned variables.

The *Stepwise Adaptation of Weights Evolutionary Algorithm* has only a single genetic operator: a mutation operator. The mutation operator implements a simple swap of the values of two randomly chosen variables. It takes a single parent and produces a single child. In [17], other mutation operators, and a number of crossover operators were tried without significant improvement of the performance. The *Stepwise Adaptation of Weights Evolutionary Algorithm* uses a biased ranked parent selection operator and a replace worst survivor selection operator.

7.8.1 *SAWEA* Characteristics and Parameter Setup

Table 7.56 shows the characteristics table of the *Stepwise Adaptation of Weights Evolutionary Algorithm*. The *Stepwise Adaptation of Weights Evolutionary Algorithm* uses a steady state evolutionary model, a biased ranking parent selection operator, and a replace worst survivor selection operator, explained in Chapter 5. The *Stepwise Adaptation of Weights Evolutionary Algorithm* uses a permutation of variables representation for the decoder. It has no crossover operator and uses a simple swap operator as a mutation operator. The fitness function of the *Stepwise Adaptation of Weights Evolutionary Algorithm* is the f_2 fitness function (see Chapter 5) with the addition of the stepwise adaptation of weights mechanism, explained in the previous section.

Table 7.57 shows the parameter table of the *Stepwise Adaptation of Weights Evolutionary Algorithm*. The *Stepwise Adaptation of Weights Evolutionary Algorithm* has a population of 10 individuals (Population Size), from which 10 parents are selected using the biased ranking parent selection operator with a bias of 1.5 (Ranking Bias). The weights of the stepwise adaptation of weights mechanism are updated every 25 generations of the algorithm (Update Interval). Weights are increased by adding 1 (Δw). Since *Stepwise Adaptation of Weights Evolutionary Algorithm* has no crossover operator, no crossover rate is needed. Also, the swap mutation operator has no parameter. The experiments of the *Stepwise Adaptation of Weights Evolutionary Algorithm* are terminated after $100,000$ fitness evaluations (Maximum Number of Evaluations).

7.8.2 *SAWEA* Experimental Results

Table 7.58 shows that the *SAWEA* has a *SR* of 1.0 for all but two density-tightness combinations in the solvable region. The *SAWEA* has reasonable *SR* in the mushy region as well. Table 7.59 shows that for most of the solvable region, the *SAWEA* will find a solution in the first generation. In the mushy region, the *AES* is low as well. There has been some discussion about whether the fitness evaluations used for calculating the weights should be counted at all. Since the calculation of the fitness value is nothing

SAWEA	
Evolutionary Model	Steady State
Representation	Permutation of Variables
Objective Function	f_2 with SAW mechanism
Crossover operator	None
Mutation operator	Swap
Parent Selection	Biased Ranking
Survivor Selection	Replace Worst
Other Functions	Decoder

Table 7.56: Characteristics of the *SAWEA*.

SAWEA	
Population Size	10
Selection Size	10
Maximum Number of Evaluations	$100,000$
Update Interval	25
Δw	1
Ranking Bias	1.5

Table 7.57: Parameters of the *SAWEA*.

more then calculating the sum of the weights for the violated constraints or unassigned variables in the individual, with a little extra storage, counting this as a full fitness evaluation seems unfair. However, if the weights are calculated for violated constraints, a list of violated constraints has to be stored, while if the weights are calculated for unassigned variables, the decoded candidate solution has to be stored. When the re-calculation of a sum argument is to be maintained therefore, the space complexity of the algorithm is increased by the extra storage space needed. Since none of the measures used measures the space complexity of an algorithm, we decided that to reflect this extra complexity, the computational complexity of the algorithm should be proportionally increased. Therefore we decided to count the re-calculation of the weights for all individuals in the population as a fitness evaluation. This allows for no “tricks” to reduce the computational complexity of the algorithm at the cost of the space complexity of the algorithm. Also, by counting all fitness evaluations equally, different values for the update interval parameter have an effect on the efficiency of the algorithm as shorter update interval parameter values result in more fitness evaluations than longer ones. Since each fitness evaluation in the *SAWEA* uses a number of conflict checks as well, this also has an effect on the *CC* measure. Overall, we believe that this allows a fairer comparison with the other algorithms in the inventory. For those who are interested in the *AES* and *CC* measures which do not count the fitness evaluations used for re-calculating the fitness values of the individuals at the weight updates, subtract one divided by the update interval parameter fitness evaluations and conflict checks from the *AES* and *CC* measures for a rough estimate. Table 7.60 shows that the *SAWEA* uses many *CC* even for solving the CSP instances in the solvable region. Since conflict checks are only used in the objective function of the *SAWEA*, this can only be explained by the fact that the decoding of an individual is expensive.

The *UIC* plots for the *SAWEA* in Figure 7.23 show that it searches through a large portion of the search space, even though that search space is limited by the use of the permutation representation. The *MBF/MCE* plots in 7.24 show that the behaviour of the *MBF* and the *MCE* is very different during the run. The reason for this is the difference between the SAW objective function with its stepwise adaptation of weights mechanism and the way the *MCE* is calculated. Weights in the SAW objective function can only increase which results in, increasing fitness values of the individuals during the run of the *SAWEA*. The *MCE* shows a more erratic behaviour. This is because the relationship between the decoder and the fitness value of the individual. The evolutionary part of the *SAWEA* evolves permutations for the decoder to use, but a small change in the individual can lead to a large difference in the fitness value of the individual after it has been decoded. The champion error, even when averaged, can therefore be very different from one generation to the next. Overall however, we see a downward trend in the *MCE* during the run, even though there is much oscillation in the plots. For density-tightness combination (0.1,0.9), the *MBF/MCE* plot shows that the *MCE* oscillates between champion individuals which at one interval have a fitness value of one and at the next interval a fitness value of two. Which of these individuals has the best fitness value depends on the weights of the variables that are unassigned. The oscillations in the other *MBF/MCE* plots are caused by this behaviour as well.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
0.4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.828
0.5	1.0	1.0	1.0	1.0	1.0	0.96	0.32	0.396	—
0.6	1.0	1.0	1.0	1.0	0.772	0.64	—	—	—
0.7	1.0	1.0	0.904	0.664	—	—	—	—	—
0.8	1.0	1.0	0.6	—	—	—	—	—	—
0.9	0.92	0.72	—	—	—	—	—	—	—

Table 7.58: *SR* of the *SAWEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	10	10	10	10	10	10	10	10	10
0.2	10	10	10	10	10	10	10	10	10
0.3	10	10	10	10	10	10	10	12	19
0.4	10	10	10	10	11	18	72	695	3547
0.5	10	10	11	15	72	699	6481	2393	—
0.6	10	10	22	108	9511	3326	—	—	—
0.7	10	22	1389	5975	—	—	—	—	—
0.8	12	336	2134	—	—	—	—	—	—
0.9	56	849	—	—	—	—	—	—	—

Table 7.59: *AES* of the *SAWEA*.

$p_1 \backslash p_2$	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.1	5219	5249	5298	5325	5359	5401	5450	5483	5518
0.2	5274	5347	5446	5514	5514	5702	5813	5902	5982
0.3	5328	5462	5611	5764	5764	5977	6185	6502	7775
0.4	5416	5632	5830	5983	5983	7457	16701	126318	645733
0.5	5511	5775	6067	6854	6854	126795	$1 \cdot 10^6$	438562	—
0.6	5631	5929	8044	22583	$2 \cdot 10^6$	603370	—	—	—
0.7	5802	8106	246762	$1 \cdot 10^6$	—	—	—	—	—
0.8	6213	60680	412326	—	—	—	—	—	—
0.9	13679	173181	—	—	—	—	—	—	—

Table 7.60: *CC* of the *SAWEA*.

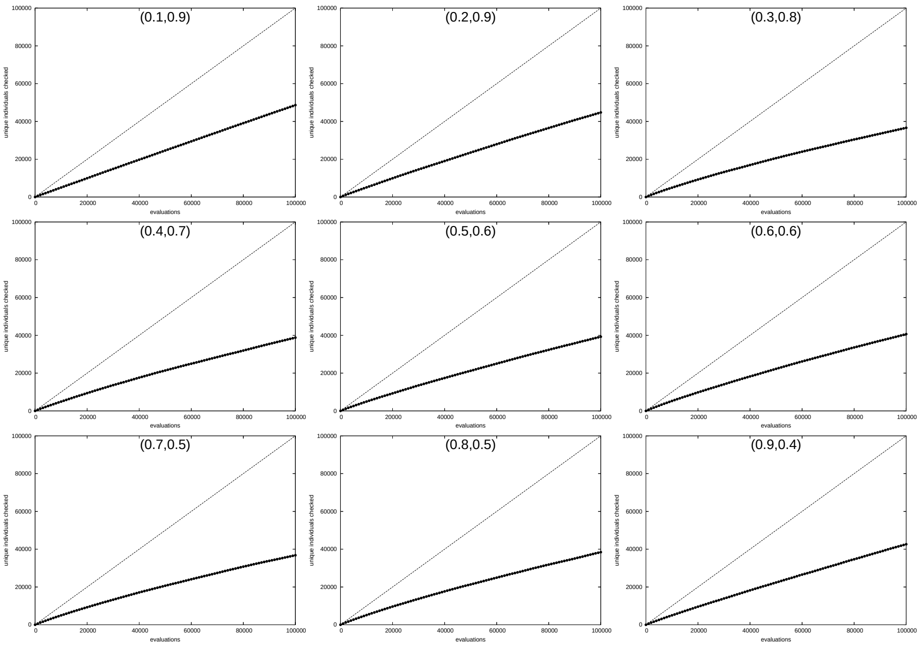

Figure 7.23: *UIC* of the *SAWEA*.

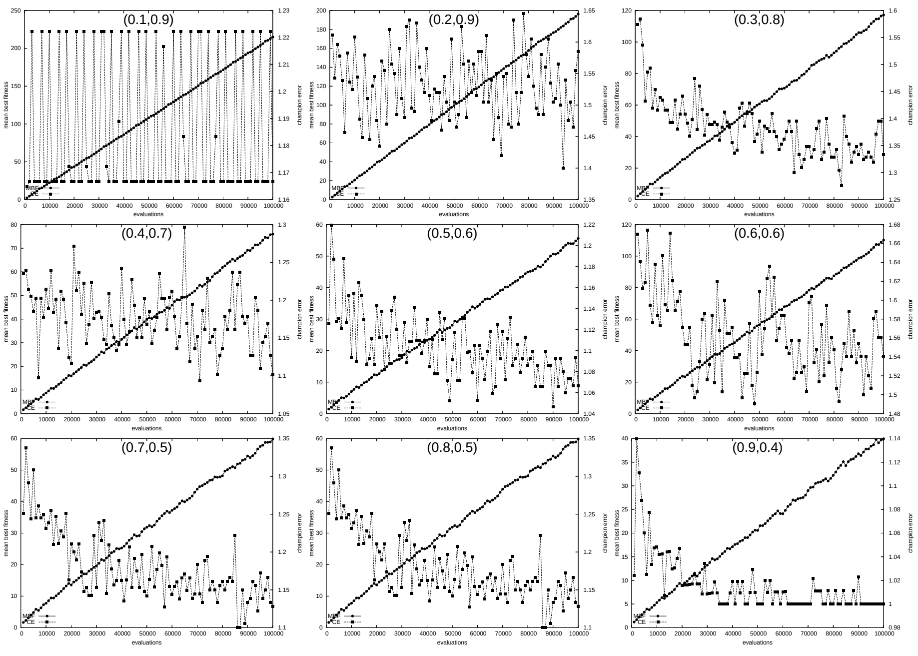

Figure 7.24: *MBF* and *MCE* of the *SAWEA*.

Chapter 8

Comparison of the Evolutionary Algorithms in the Inventory

This chapter contains a comparison of the performance of the evolutionary algorithms in the inventory given in Chapter 7. In the first section the performance of the algorithms is compared on the effectivity and efficiency measures, *SR*, *AES*, and *CC*. The second section compares the relative performance of the algorithms in the *SR*-*AES* and *SR*-*CC* planes. Statistical analysis on the effectivity measure *SR* is used to rank the performance of the algorithms in the third section. A preliminary conclusion based on the comparison is presented in the final section of the chapter.

8.1 Comparison on Effectivity and Efficiency Measures

The performance of the algorithms in the inventory is compared along the same lines as was done in Chapter 6. The performance of all algorithms is summarised in three tables, one for each performance measure: the *SR*, the *AES*, and the *CC*. The table for the *SR* measure is shown in Table 8.1. The table for the *AES* measure is shown in Table 8.2. The table for the *CC* measure is shown in Table 8.3. In each table, for each density-tightness combination, the best value is shown in bold-face.

Table 8.1 shows that the *LSEA* has the best average *SR* of all algorithms in the inventory. For density-tightness combination (0.1,0.9), the *HEA1*, the *HEA3*, the *ESPEA*, and the *LSEA* solved all CSP instances in all runs. The *ArcEA1*, the *HPEA*, and the *SAWEA* had a *SR* of 0.989, 0.98, and 0.92 respectively. These algorithms were able to solve the CSP instances for this density-tightness combination in nearly all runs. For density-tightness combination (0.2,0.9), the *LSEA* has the best *SR*: 0.988. The *HEA3* had the second best *SR* with 0.984. The other algorithms had a significantly lower *SR*. For density-tightness combination (0.3,0.8), *LSEA* solved the CSP instances in the most

	$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
HEA1	**1.0**	0.892	0.556	0.572	0.504	0.42	0.4	0.428	0.504
HEA2	0.764	0.188	0.068	0.08	0.072	0.056	0.04	0.064	0.076
HEA3	**1.0**	0.984	0.688	0.712	0.692	0.44	0.588	0.488	0.76
ArcEA1	0.988	0.688	0.368	0.384	0.312	0.284	0.22	0.24	0.3
ArcEA2	0.708	0.12	0.016	0.02	0.016	0.024	0.008	0.008	0.012
ArcEA3	0.692	0.128	0.024	0.032	0.012	0.028	0.012	0.004	0.008
CoeEA	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ESPEA	**1.0**	0.676	0.388	0.436	0.436	0.404	0.328	0.468	0.432
HPEA	0.98	0.564	0.256	0.228	0.188	0.188	0.152	0.204	0.156
LSEA	**1.0**	**0.988**	**0.812**	**0.808**	**0.924**	**0.752**	**0.776**	**0.796**	**0.936**
MIDEA	0.22	0.004	0.0	0.004	0.0	0.004	0.0	0.0	0.0
SAWEA	0.92	0.72	0.6	0.664	0.772	0.64	0.32	0.396	0.828

Table 8.1: Comparison table *SR*.

runs with a *SR* of 0.812, all other algorithms had a lower *SR* with *HEA3* having the second best *SR* of 0.688. The other density-tightness combinations in the mushy region show a comparable *SR* distribution, although sometimes *HEA3* had the second highest *SR* while for other density-tightness combinations the *SAWEA* had the second highest *SR*. Overall, the *SR* of *HEA3* and *SAWEA* are fairly close to each other but not as high as *LSEA*.

The comparison tables for the *AES* and *CC* measures (Tables 8.2 and 8.3) do not show such a clear-cut advantage of one algorithm. Not only are the differences between the *AES* and *CC* measures more varied, different algorithms throughout the mushy region use less *AES* and *CC*. Overall, the *ArcEA2* has the lowest *AES* and *CC*, however, the *SR* of the *ArcEA2* is relatively low, making both measures less accurate. The *LSEA* with the highest *SR* has the most accurate *AES* and *CC* measures.

From all three tables it is clear that the *CoeEA* has the worst performance of all algorithms in the inventory. If fails to solve a single CSP instance in the mushy region in all its runs. The *MIDEA* also has poor performance. It has a low *SR* throughout the mushy region and solves the CSP instances in the mushy region only for a small number of runs and then only in 4 out of 9 density-tightness combinations. For the *CC* measure, note that both the *ESPEA* and the *LSEA* use a lot more conflict checks than the other algorithms. Compared to the *HEA2*, another algorithm with high *CC* values, the *ESPEA* uses, on average, between 2.49 ((0.1,0.9)) to 57.68 ((0.9,0.4)) times as many conflict checks to find a solution. The *LSEA* uses even more conflict checks, on average, between 2.23 ((0.1,0.9)) to 575.74 ((0.5,0.6)) times as many. Although both the *ESPEA* and the *LSEA* have an above average *SR*, this comes at the price of a high *CC*.

	$(\mathbf{0.1},\ \mathbf{0.9})$	$(\mathbf{0.2},\ \mathbf{0.9})$	$(\mathbf{0.3},\ \mathbf{0.8})$	$(\mathbf{0.4},\ \mathbf{0.7})$	$(\mathbf{0.5},\ \mathbf{0.6})$	$(\mathbf{0.6},\ \mathbf{0.6})$	$(\mathbf{0.7},\ \mathbf{0.5})$	$(\mathbf{0.8},\ \mathbf{0.5})$	$(\mathbf{0.9},\ \mathbf{0.4})$
HEA1	37	**335**	3931	1448	3387	5704	1951	7603	2789
HEA2	5862	14268	13660	21876	10727	13596	14444	13596	16609
HEA3	26	419	1635	1404	2382	988	969	1258	1563
ArcEA1	279	3467	2008	4403	962	2099	2116	778	5067
ArcEA2	2804	8269	**362**	**186**	494	**186**	**218**	1953	**250**
ArcEA3	2036	4056	648	2906	**173**	8060	2720	**290**	1225
CoeEA	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*
ESPEA	997	6604	4982	6280	7928	5581	8599	5332	8365
HPEA	2727	15222	23212	20118	20224	22063	21258	20629	20945
LSEA	**13**	540	9825	5935	10124	12080	11562	11422	4097
MIDEA	413	550	*undef.*	940	*undef.*	6635	*undef.*	*undef.*	*undef.*
SAWEA	56	849	2134	5975	9511	3326	6481	2393	3547

Table 8.2: Comparison table *AES*.

8.2 Comparison on the Effectivity-Efficiency Plane

The tables in the previous section show that looking at the *SR*, *AES*, and *CC* separately does not provide us with a complete picture. We already explained that there is a relationship between the *SR* measure and the *AES* and the *CC* measures in that the *SR* influences the accuracy of the *AES* and *CC* measures. In addition to this relationship, there exists another relationship between the effectivity and efficiency measures. Ideally, an algorithm should have both a good effectivity *and* a good efficiency, i.e., a high *SR* and a low *AES* and *CC*. From the tables in the previous section it is clear that this is not the case, the *LSEA* has the best overall *SR* of all algorithms in the inventory but a high *AES* and *CC*.

To compare the effectivity-efficiency relationship of each algorithm we use plots with on the x-axis the *SR* of the algorithm and on the y-axis either the *AES* or the *CC* performance. In total two sets of nine plots, one for each density-tightness combination in the mushy region are used, one set for the *SR-AES* relationship and one for the *SR-CC* relationship. The *SR* measure already has a range between 0.0 and 1.0, but we normalise the *AES* and *CC* measures to this range as well. Figure 8.1 shows the first set of plots for the *SR-AES* relationship. Figure 8.2 shows the second set of plots for the *SR-CC* relationship. Because of the large spread between the *CC* values for the algorithms we used a logarithmic scale on the y-axis in the Figure 8.2. The *CoeEA* has a *SR* of 0.0 for all density-tightness combinations in the mushy region and an undefined *AES* and *CC* measure, and this algorithm is not represented in the plots. The same applies for the *MIDEA* for 5 out of the 9 density-tightness combinations. The plots show the other algorithms as a dot labelled with the abbreviation of the algorithm.

Two methods can be used to determine the order of the *SR-AES* and the *SR-CC* rela-

	$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
HEA1	**13**	167	2015	761	1721	2947	1043	4065	1502
HEA2	3980	9752	9405	15153	7481	9538	10205	8456	11885
HEA3	24	621	2489	2110	3647	1493	1472	1933	2405
ArcEA1	67	866	523	1205	261	588	569	220	1513
ArcEA2	68	352	**24**	**15**	**24**	**254**	**41**	**51**	**48**
ArcEA3	73	252	60	354	26	1479	597	67	333
CoeEA	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*	*undef.*
ESPEA	9918	118774	139361	225890	364466	301103	549994	383535	685535
HPEA	85	718	1255	1152	1102	1336	1535	1503	1420
LSEA	8893	300212	$5 \cdot 10^6$	$3 \cdot 10^6$	$4 \cdot 10^6$	$5 \cdot 10^6$	$5 \cdot 10^6$	$4 \cdot 10^6$	$2 \cdot 10^6$
MIDEA	56	**74**	*undef.*	127	*undef.*	897	*undef.*	*undef.*	*undef.*
SAWEA	14	173	412	1111	1732	603	1170	439	646

Table 8.3: Comparison table *CC*.

tionships of the algorithms in the plots.

In the first method we partition each plots into four quadrants, numbered one to four, clockwise. The first quadrant then includes algorithms with a *SR* of 0.5 or more and an *AES* or *CC* of more than half of the maximum found. In quadrant 2 the algorithms with a *SR* of 0.5 or more and an *AES* or *CC* of less then half the maximum can be found. In the third quadrant the algorithms with a *SR* of less then 0.5 and less then half the maximum *AES* can be found. In the fourth quadrant the algorithms with a *SR* of 0.5 and an *AES* and *CC* of more then half the maximum can be found. The algorithms with a better *SR*-*AES* or *SR*-*CC* relationship can thus be found in quadrant 2 (bottom-right) while the algorithms with a worse relationship are located in the fourth quadrant (top-left). Quadrants can be further subdivided for a more fine-grained determination of the ordering. The quadrant method is slightly more complicated for the plots in Figure 8.2 because of the logarithmic scale of the y-axis, resulting in quadrants that are not equal in height.

The second method to determine the order of the *SR*-*AES* and the *SR*-*AES* relationships of the algorithms involves moving a line at an angle to the x-axis from the bottom-left corner to the top-right corner of each plot. The dot of the algorithm that is first crossed by the line is then the algorithm with the best *SR*-*AES* or *SR*-*CC* relationship. The one that is crossed last is the algorithm with the worst relationship. The angle to the x-axis of the plot is determined by the (relative) weight applied to the importance of the performance measure. If the *SR* is equal in importance to either the *AES* and the *CC* measure, this angle is 45 degrees. The angle is decreased when the importance of the *SR* in increased and the angle is increased otherwise. The line can be described by the following formula: $y = \frac{w_{SR}}{w_{AES}} \cdot x + a$ for the *SR*-*AES* relationship and $y = \frac{w_{SR}}{w_{CC}} \cdot x + a$ for the *SR*-*CC* relationship where w_{SR} is the relative weight of the *SR* measure, w_{AES} the relative weight of the *AES* measure, w_{CC} the relative weight of the *CC* measure, and a is used to move the line. Here we assume equal weight of the two performance measures ($w_{SR} = w_{AES} = w_{CC}$). Again the method is slightly more complicated by the logarithmic scale of the y-axis in Figure 8.2 as the lines will show up in the plots as logarithmic curves.

Using the first method to order the *SR-AES* relationship in Figure 8.1 shows that for density-tightness combination (0.1,0.9) most algorithms can be found in the second (bottom-right) quadrant. Only the *HEA2* and the *MIDEA* are outside this quadrant. For density-tightness combination (0.2,0.9), the *LSEA*, the *HEA3*, the *HEA1*, the *SAWEA*, the *ArcEA1*, and the *ESPEA* lie in the second quadrant. The plots for density-tightness combinations (0.3,0.8) and (0.4,0.7) show that the *LSEA*, the *HEA3*, the *HEA1*, and the *ESPEA* lie in the second quadrant. For density-tightness combinations (0.5,0.6) and (0.9,0.4), the *LSEA*, the *SAWEA*, and the *HEA3* lie in the second quadrant while for density-tightness combinations (0.6,0.6) and (0.7,0.5), the *LSEA* and the *HEA3* lie in the second quadrant. For the remaining density-tightness combination, (0.8,0.5), only the *LSEA* lies in the second quadrant. Overall, both the *LSEA* and the *HEA3* have both a high *SR* and a low *AES*. The *HPEA* and the *HEA2* often lie in the fourth quadrant and the *ArcEA2* and the *ArcEA3* lie often in the third quadrant (bottom-left).

Using the first method to order the *SR-CC* relationship in Figure 8.2 is more complicated because of the logarithmic scale of the y-axis. Nevertheless, the plots show a very different relationship between the *SR* and the *CC* than was seen in Figure 8.1. In Figure 8.1 the *LSEA* and to a lesser extend the *ESPEA* had relatively low *AES* while in Figure 8.2 both algorithms can always be found towards the top of the plots. Relative to the *CC* of the other algorithms therefore, these two algorithms have a high *CC* in relation to a high *SR*. Because the y-axis of the plots in Figure 8.2 is in logarithmic scale, this difference is large, reflecting our earlier observations in the previous section.

Figures 8.1 and 8.2 indicate a different relationship between the *SR* and the *AES* of the algorithms then for the *SR* and the *AES*. Although the *HEA3* and to a lesser extend the *HEA1* and the *SAWEA* were located near the bottom-left corners of both graphs, the *LSEA* and the *ESPEA* were located in the bottom-left corner in Figure 8.1 and in the top-left corner in Figure 8.2. This is an indication of the large amount of (hidden) work that the *LSEA* and the *ESPEA* need to do to attain the high *SR* they have. In contrast the *HEA3* also has a good *SR* but needs much less conflict checks to attain this.

The use of a moving line in the second method of determining the order of the relationship between *SR-AES* and *SR-CC* shows us that the order can also be determined by the ratio of the *SR* and the *AES* or *CC* multiplied by the ratio of the weights for these measures, as in the formula: $o = \frac{w_{SR}}{w_{AES}} \cdot \frac{SR}{AES}$, where o-values determine the relative order of the algorithms. The meaning of the w_{SR} and w_{AES} variables has been explained above. The formula signifies the rewriting of the previous formula in order to find a when x and y are known. When we assume equal importance of *SR* to *AES* and *CC*, the values for o in Table 8.4 for the *SR-AES* relationship and Table 8.5 for the *SR-CC* relationship can be calculated. As in Figures 8.1 and 8.2 we used the normalised values of the *AES*, and the *CC*. Based on these o-values, we can determine the order of the algorithms based on the two relationships. The orders for each density-tightness combination in the mushy region for the *SR-AES* and the *SR-CC* relationship are shown in Table 8.6

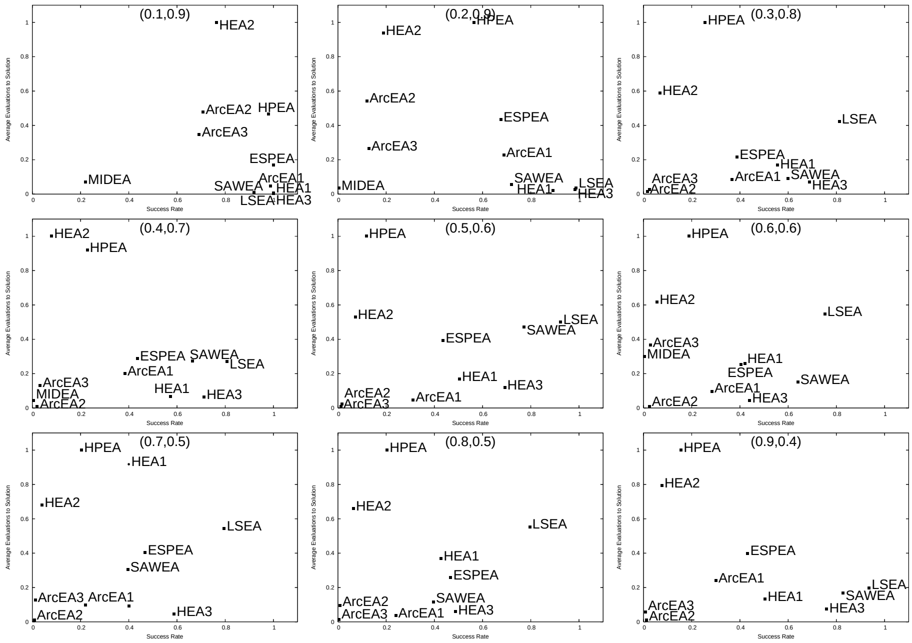

Figure 8.1: Algorithm distribution on the *SR*-*AES* plane.

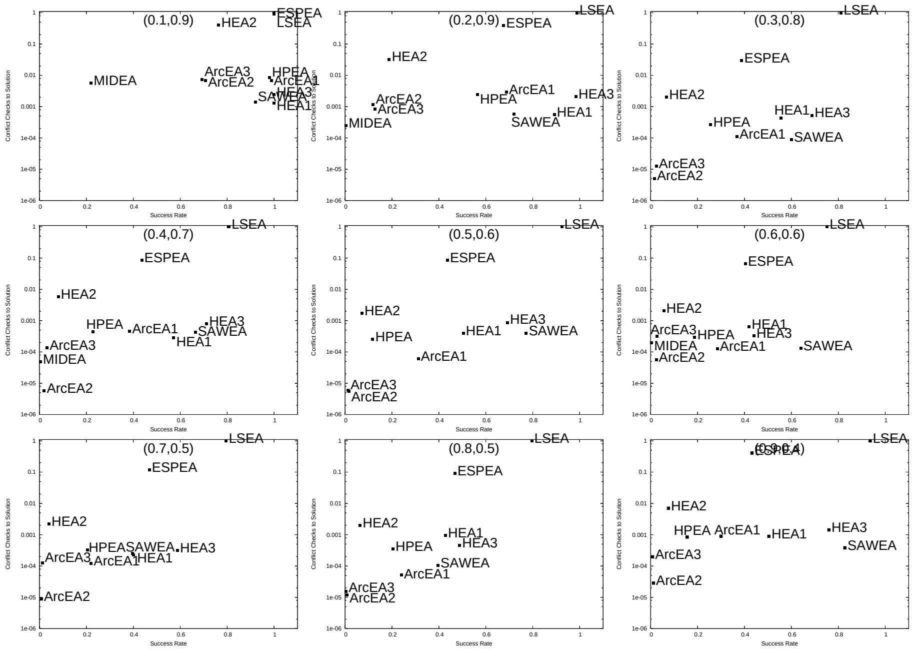

Figure 8.2: Algorithm distribution on the *SR*-*CC* plane.

and 8.7 respectively.

Tables 8.6 and 8.7 show an entirely different picture for the *SR*-*AES* and the *SR*-*CC* relationships. As already shown in the previous section, the *LSEA* and the *ESPEA* have reasonably low *AES* values for the experiments and in Table 8.6 both algorithms can be found near the top of order for all density-tightness combinations. At the same time, both algorithms also have high *CC* values and as a consequence can be found near the bottom of the ordering in 8.7. At the same time, an algorithm like the *SAWEA* which has about average *SR* but both low *AES* and *CC* is found in the top of the orderings of both Tables 8.6 and 8.7.

A word of caution for the interpretation of these tables is necessary. The *SR* of an algorithm, that is, the ability of the algorithm to solve the CSP, is clearly more important than the efficiency of the algorithm. Therefore, the assumption that the importance of both the effectivity *and* the efficiency is equal is probably not correct. However, without extra guidance upon the relative importance of these measures, it is not possible to set it with any degree of certainty. Furthermore, there is the implicit assumption that all measures upon which the calculations of the o-values are based are accurate. This is not the case. With a lower *SR*, the accuracy of the *AES* and *CC* measures is also lower. Taken together, the comparison on the effectivity-efficiency plane should be taken as guidance towards an ordering of the algorithms more than experimental fact. Taken as such, however, they are useful in at least quantifying the relative advantages of one algorithm over another based on the relationship between the different performance measures. This ties in with the use of a restart strategy for evolutionary algorithms and the use of the relationship between the effectivity and the efficiency measures to estimate the duration of the experiments and the number of restarts needed during the experiments based on the *SR* and the *AES* and *CC* measures to attain a *SR* of 1.0. We feel, however, that a further discussion of this topic (which involves a number of other factors not discussed so far) falls outside the scope of the thesis (see [40] for more information).

8.3 Ranking of the Evolutionary Algorithms in the Inventory

Although Tables 8.6 and 8.7 give an indication of a ranking of the algorithms according to their relative performance in the *SR*-*AES* and *SR*-*CC* planes, the drawbacks to the ranking mechanism given above make these rankings tentative. Especially the inability to categorically state the relative importance of the effectivity measure (*SR*) to the effectivity measures (*AES* and *CC*) has the potential to skew the rankings.

Statistical analysis on only the effectivity measure (*SR*) is used to rank the algorithms more accurately. By basing the analysis on the *SR* measure only, we acknowledge that the effectivity of an algorithm is more important than the efficiency of an algorithm. The choice of only analysing the *SR* measure is also based on the fact that this measure takes the whole results sample into account while the *AES* and the *CC* measures are calculated only over the successful runs of an algorithm. This makes the *SR* measure

	(0.1, 0.9)	(0.2, 0.9)	(0.3, 0.8)	(0.4, 0.7)	(0.5, 0.6)	(0.6, 0.6)	(0.7, 0.5)	(0.8, 0.5)	(0.9, 0.4)
HEA1	158.43	44.62	3.28	8.64	3.01	1.62	4.36	1.16	3.78
HEA2	0.76	0.20	0.12	0.08	0.14	0.09	0.06	0.10	0.10
HEA3	225.46	35.75	9.77	11.09	5.88	9.83	12.9	8.00	10.18
ArcEA1	20.76	3.02	4.25	1.91	6.56	2.99	2.21	6.36	1.24
ArcEA2	1.48	0.22	1.03	2.35	0.66	2.85	0.78	0.08	1.01
ArcEA3	1.99	0.48	0.86	0.24	1.40	0.08	0.09	0.28	0.14
CoeEA	-	-	-	-	-	-	-	-	-
ESPEA	5.88	1.56	1.81	1.52	1.11	1.60	1.16	1.81	1.08
HPEA	2.11	0.56	0.26	0.25	0.12	0.19	0.20	0.20	0.16
LSEA	450.92	27.85	1.92	2.98	1.85	1.37	1.46	1.44	4.79
MIDEA	3.12	0.11	-	0.09	-	0.01	-	-	-
SAWEA	96.30	12.91	6.53	2.43	1.64	4.25	1.30	3.41	4.89

Table 8.4: o-values for the algorithms on the *SR*-*AES* plane.

	(0.1, 0.9)	(0.2, 0.9)	(0.3, 0.8)	(0.4, 0.7)	(0.5, 0.6)	(0.6, 0.6)	(0.7, 0.5)	(0.8, 0.5)	(0.9, 0.4)
HEA1	763.0	1603.5	1300.8	1985.6	1261.4	651.7	1792.5	450.6	567.0
HEA2	1.9	5.8	34.1	14.0	41.5	26.9	18.3	32.4	10.8
HEA3	413.3	475.7	1303.1	891.4	817.3	1347.7	1867.0	1080.4	534.0
ArcEA1	146.3	238.5	3317.1	841.8	5148.8	2208.7	1807.1	4668.5	335.1
ArcEA2	103.3	102.4	3142.9	3522.1	2871.4	432.1	912.0	671.3	422.5
ArcEA3	94.0	152.5	1885.8	238.8	1987.9	86.6	94.0	255.5	40.6
CoeEA	-	-	-	-	-	-	-	-	-
ESPEA	1.0	1.7	13.1	5.1	5.2	6.1	4.0	5.2	1.1
HPEA	114.4	235.8	961.7	522.8	461.2	643.5	621.2	580.9	185.6
LSEA	1.1	1.0	0.8	0.8	0.9	0.8	0.8	0.8	0.9
MIDEA	39.0	16.2	-	83.2	-	20.4	-	-	-
SAWEA	651.7	1249.4	6865.5	1578.8	1919.8	4853.6	1581.9	3860.4	2165.8

Table 8.5: o-values for the algorithms on the *SR*-*CC* plane.

$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
LSEA	*HEA1*	*HEA3*	*HEA3*	*ArcEA1*	*HEA3*	*HEA3*	*HEA3*	*HEA3*
HEA3	*HEA3*	*SAWEA*	*HEA1*	*HEA3*	*SAWEA*	*HEA1*	*ArcEA1*	*SAWEA*
HEA1	*LSEA*	*ArcEA1*	*LSEA*	*HEA1*	*ArcEA1*	*ArcEA1*	*SAWEA*	*LSEA*
SAWEA	*SAWEA*	*HEA1*	*SAWEA*	*LSEA*	*ArcEA2*	*LSEA*	*ESPEA*	*HEA1*
ArcEA1	*ArcEA1*	*LSEA*	*ArcEA2*	*SAWEA*	*HEA1*	*SAWEA*	*LSEA*	*ArcEA1*
ESPEA	*ESPEA*	*ESPEA*	*ArcEA1*	*ArcEA3*	*ESPEA*	*ESPEA*	*HEA1*	*ESPEA*
MIDEA	*HPEA*	*ArcEA2*	*ESPEA*	*ESPEA*	*LSEA*	*ArcEA2*	*ArcEA3*	*ArcEA2*
HPEA	*ArcEA3*	*ArcEA3*	*HPEA*	*ArcEA2*	*HPEA*	*HPEA*	*HPEA*	*HPEA*
ArcEA3	*ArcEA2*	*HPEA*	*ArcEA3*	*HEA2*	*HEA2*	*ArcEA3*	*HEA2*	*ArcEA3*
ArcEA2	*HEA2*	*HEA2*	*MIDEA*	*HPEA*	*ArcEA3*	*HEA2*	*ArcEA2*	*HEA2*
HEA2	*MIDEA*	*MIDEA*	*HEA2*	*MIDEA*	*MIDEA*	*MIDEA*	*MIDEA*	*MIDEA*
CoeEA	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*

Table 8.6: Order of the algorithms on the *SR-AES* plane.

$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
HEA1	*HEA1*	*SAWEA*	*ArcEA2*	*ArcEA1*	*SAWEA*	*HEA3*	*ArcEA1*	*SAWEA*
SAWEA	*SAWEA*	*ArcEA1*	*HEA1*	*ArcEA2*	*ArcEA1*	*ArcEA1*	*SAWEA*	*HEA1*
HEA3	*HEA3*	*ArcEA2*	*SAWEA*	*ArcEA3*	*HEA3*	*HEA1*	*HEA3*	*HEA3*
ArcEA1	*ArcEA1*	*ArcEA3*	*HEA3*	*SAWEA*	*HEA1*	*SAWEA*	*ArcEA2*	*ArcEA2*
HPEA	*HPEA*	*HEA3*	*ArcEA1*	*HEA1*	*HPEA*	*ArcEA2*	*HPEA*	*ArcEA1*
ArcEA2	*ArcEA3*	*HEA1*	*HPEA*	*HEA3*	*ArcEA2*	*HPEA*	*HEA1*	*HPEA*
ArcEA3	*ArcEA2*	*HPEA*	*ArcEA3*	*HPEA*	*ArcEA3*	*ArcEA3*	*ArcEA3*	*ArcEA3*
MIDEA	*MIDEA*	*HEA2*	*MIDEA*	*HEA2*	*HEA2*	*HEA2*	*HEA2*	*HEA2*
HEA2	*HEA2*	*ESPEA*	*HEA2*	*ESPEA*	*MIDEA*	*ESPEA*	*ESPEA*	*ESPEA*
LSEA	*ESPEA*	*LSEA*	*ESPEA*	*LSEA*	*ESPEA*	*LSEA*	*LSEA*	*LSEA*
ESPEA	*LSEA*	*MIDEA*	*LSEA*	*MIDEA*	*LSEA*	*MIDEA*	*MIDEA*	*MIDEA*
CoeEA	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*	*CoeEA*

Table 8.7: Order of the algorithms on the *SR-CC* place.

intrinsically more accurate.

The following symbols are used to denote the relative performance of two algorithms: $A_1 > A_2$ indicates that algorithm A_1 has a higher *SR* than algorithm A_2, $A_1 \gtrsim A_2$ indicates that algorithm A_1 has higher or similar *SR* than algorithm A_2, $A_1 \backsimeq A_2$ indicates that algorithm A_1 has approximately similar *SR* than algorithm A_2, and $A_1 \gg A_2$ indicates that algorithm A_1 has far higher *SR* than algorithm A_2. The symbols are transitive in an ordering of more than two algorithms.

The statistical analysis uses the two sample t-test to compare the performance of two algorithms. Only the *SR* measure will be considered for the statistical analysis. The same three hypotheses are used for the two sample t-test as were used in Chapter 6:

$$H_0 : \overline{SR}_{A_1} = \overline{SR}_{A_2} \tag{8.1}$$

$$H_{a_1} : \overline{SR}_{A_1} \neq \overline{SR}_{A_2} \tag{8.2}$$

$$H_{a_2} : \overline{SR}_{A_1} > \overline{SR}_{A_2} \tag{8.3}$$

where A_1 stands for the first algorithm and A_2 for the second. For a full analysis, t-tests for all algorithm combinations have to be done. We reduce the number of t-tests needed by first ordering the algorithms based to the *SR* results from Table 8.1 and then re-ordering the algorithms them when necessary. Eventually, the following ranking was found:

$$\begin{aligned} &\textit{LSEA} > \textit{HEA3} \gtrsim \textit{HEA1} \gtrsim \textit{ESPEA} \gtrsim \ldots \\ &\qquad \ldots \gtrsim \textit{ArcEA1} \gtrsim \textit{SAWEA} \gtrsim \textit{HPEA} > \textit{HEA2} > \ldots \\ &\qquad\qquad \ldots > \textit{ArcEA2} \backsimeq \textit{ArcEA3} \gg \textit{MIDEA} > \textit{CoeEA} \end{aligned} \tag{8.4}$$

The results of the t-tests for every algorithm pair in the ranking, 11 in total, are given in Table 8.8. A t-test for every density-tightness combination in the mushy region was done. The t-test results for every algorithm pair are shown in three lines. The first gives the p-value for the t-test on h_0 and h_{a_1}, the second gives the p-value for the t-test on h_0 and h_{a_2}. The interpretation of the two p-values is given on the third line, using the symbols $=$, when the *SR* results of both algorithms are equal, $>$ when the *SR* results of algorithm A_1 are better than those of algorithm A_2, and $<$ when the *SR* results of algorithm A_1 are worse than those of algorithm A_2. The symbols $\gtrsim$ and $\lesssim$ are used when the difference between the *SR* results are similar but better or worse for algorithm A_1 than for algorithm A_2 respectively. The p-values are interpreted as follows: when the p-value of a t-test is low, say below 0.5, than the possibility of h_0 being correct is also low, and therefore the possibility of the alternative hypothesis, either h_{a_1} or h_{a_2}, being correct is high. The opposite is true when the p-value is high. Therefore, when the p-value of a t-test is high, there is no significant difference between the *SR* results of the two compared algorithms. When it is low there is a significant difference between the *SR* results of the algorithm. For the second t-test, between h_0 and h_{a_2}, a low p-value means that the *SR* results of the first algorithm are significantly better than the *SR* results of the second algorithm. No t-test is possible when there are no results for both algorithms (a *SR* of 0.0). When both algorithms solve all CSP instances in all runs, there is no difference between the *SR* results of the two algorithms, and no p-value can be calculated. In both cases the absence of a p-value is interpreted with an $=$ symbol.

The t-test results in Table 8.8 for each algorithm pair is discussed below:

LSEA $>$ ***HEA3*** The *LSEA* has better *SR* results than the *HEA3* for density-tightness combinations (0.3,0.8) to (0.9,0.4). Both algorithms solved all CSP instances in all runs for density-tightness combination (0.1,0.9). For density-tightness combination (0.2,0.9) the difference between the two algorithms is not as large, there is a 0.70 probability of the *SR* results of the two algorithms being equal and a 0.65 probability of the *SR* results of the *LSEA* being better than the *SR* results of the *HEA3*.

HEA3 $\gtrsim$ ***HEA1*** Both the *HEA3* and the *HEA1* solved all CSP instances in all runs for density-tightness combination (0.1,0.9). For all other density-tightness combinations with the exception of (0.6,0.6) the *HEA3* has better *SR* results than the *HEA1*. For density-tightness combination (0.6,0.6), the probability of the *HEA3* having equal *SR* results than the *HEA1* is 0.65, the probability of the *HEA3* having better *SR* results for that density-tightness combination is 0.67.

HEA1 $\gtrsim$ ***ESPEA*** Both the *HEA1* and the *ESPEA* solved all CSP instances in all runs for density-tightness combination (0.1,0.9). For all other density-tightness combinations with the exception of (0.6,0.6), the *HEA1* has better *SR* results than the *ESPEA*. For density-tightness combination (0.6,0.6), the probability of the *HEA1* having equal *SR* results than *ESPEA* is 0.72 while the probability of the *HEA1* having better *SR* results than the *ESPEA* is 0.64.

ESPEA $\gtrsim$ ***ArcEA1*** The *ESPEA* has better *SR* results than the *ArcEA1* for all density-tightness combinations in the mushy region except for (0.2,0.9) and (0.3,0.8), where the probability of the *ESPEA* having equal *SR* results with the *ArcEA1* is 0.77 and 0.65 respectively while the probability of the *ESPEA* having better *SR* results is 0.29 and 0.68 respectively.

ArcEA1 $\gtrsim$ ***SAWEA*** The *ArcEA1* has better *SR* results than the *SAWEA* for density-tightness combinations (0.1,0.9) to (0.3,0.8) but worse *SR* results for all other density-tightness combinations. The probabilities for the better *SR* results in the first three density-tightness combinations are however higher than the probabilities for the worse *SR* results in the other density-tightness combinations. Also, when the *ArcEA1* was compared with the *HPEA*, it showed clearly better *SR* results for all density-tightness combinations (not shown in the table). This indicates that the position where the *ArcEA1* is ranked is correct although for some density-tightness combinations in the mushy region the *SAWEA* is actually better than the *ArcEA1*.

	$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
LSEA $>$ *HEA3*	-	0.70	0.0	0.01	0.0	0.0	0.0	0.0	0.0
	-	0.35	0.0	0.01	0.0	0.0	0.0	0.0	0.0
	$=$	$\gtrsim$	$>$	$>$	$>$	$>$	$>$	$>$	$>$
HEA3 $\gtrsim$ *HEA1*	-	0.0	0.0	0.0	0.0	0.65	0.0	0.18	0.0
	-	0.0	0.0	0.0	0.0	0.33	0.0	0.09	0.05
	$=$	$>$	$>$	$>$	$>$	$\gtrsim$	$>$	$>$	$>$
HEA1 $\gtrsim$ *ESPEA*	-	0.0	0.0	0.0	0.13	0.72	0.09	0.37	0.11
	-	0.0	0.0	0.0	0.06	0.36	0.05	0.82	0.05
	$=$	$>$	$>$	$>$	$>$	$\gtrsim$	$>$	$<$	$>$
ESPEA $\gtrsim$ *ArcEA1*	0.08	0.77	0.65	0.24	0.0	0.0	0.01	0.0	0.0
	0.04	0.61	0.32	0.12	0.0	0.0	0.0	0.0	0.0
	$>$	$=$	$\gtrsim$	$>$	$>$	$>$	$>$	$>$	$>$
ArcEA1 $\gtrsim$ *SAWEA*	0.0	0.0	0.0	0.0	0.04	0.0	0.12	0.19	0.0
	0.0	0.0	0.0	1.0	0.98	1.0	0.94	0.91	1.0
	$>$	$>$	$>$	$<$	$<$	$<$	$<$	$<$	$<$
SAWEA $\gtrsim$ *HPEA*	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.02	0.0
	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.01	0.0
	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$
HPEA $>$ *HEA2*	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$
HEA2 $>$ *ArcEA2*	0.16	0.04	0.0	0.0	0.0	0.07	0.02	0.0	0.0
	0.08	0.02	0.0	0.0	0.0	0.03	0.01	0.0	0.0
	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$
ArcEA2 $\simeq$ *ArcEA3*	0.70	0.79	0.52	0.40	0.70	0.78	0.65	0.56	0.65
	0.35	0.61	0.74	0.80	0.35	0.61	0.67	0.28	0.33
	$\gtrsim$	$\gtrsim$	$\lesssim$	$\lesssim$	$\gtrsim$	$\lesssim$	$\lesssim$	$\gtrsim$	$\gtrsim$
ArcEA3 $\gg$ *MIDEA*	0.0	0.0	0.01	0.02	0.08	0.09	0.08	0.32	0.16
	0.0	0.0	0.01	0.01	0.04	0.05	0.04	0.16	0.08
	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$
MIDEA $>$ *CoeEA*	0.0	0.0	-	0.32	-	0.16	-	-	-
	0.0	0.16	-	0.16	-	0.08	-	-	-
	$>$	$>$	$=$	$>$	$=$	$>$	$=$	$=$	$=$

Table 8.8: t-test results for the ranking of the EAs in the inventory.

SAWEA* $\gtrsim$ *HPEA The *SAWEA* has better *SR* results than the *HPEA* for density-tightness combination (0.4,0.7) to (0.9,0.4) $(0.9, 0.4)$ but worse *SR* results for density-tightness combinations (0.1,0.9) to (0.3,0.8). When the *SAWEA* was compared with the *HEA2* it showed better *SR* results for all density-tightness combinations (not shown in the table), indicating that its position in the ranking is correct, even though for some density-tightness combinations in the mushy region, the *HPEA* actually has better *SR* results than the *SAWEA*. The differences between the *ArcEA1*, the *SAWEA*, and the *HPEA* are more complex than can be expressed through statistical tests between two algorithms. For some density-tightness combinations one algorithm has the better *SR* results while for other density-tightness combinations another algorithm performs best. The ranking given for these three algorithms therefore is less accurate than for the other algorithms. It is however the best interpretation that can be given using these measures.

HPEA* $>$ *HEA2 The *HPEA* has better *SR* results than the *HEA2* for all density-tightness combinations in the mushy region.

HEA2* $>$ *ArcEA2 The *HEA2* has better *SR* results than the *ArcEA2* for all density-tightness combinations in the mushy region.

ArcEA2* $\simeq$ *ArcEA3 The difference between the *SR* results of the *ArcEA2* and the *ArcEA3* are small for all density-tightness combinations in the mushy region. For density-tightness combinations (0.1,0.9), (0.2,0.9), (0.5,0.6), (0.8,0.5), and (0.9,0.4), the probability of the *HPEA* having better *SR* results than the *HEA2* is higher than for the other density-tightness combinations. We conclude that the *SR* results over the whole mushy region for the *ArcEA2* and the *ArcEA3* were approximately equal, even though there were local differences. This result does not come as a surprise since the only difference between the two algorithms is the adaptability of the arc-crossover operator in the *ArcEA3*.

ArcEA3* $\gg$ *MIDEA The *ArcEA3* has better *SR* results then the *MIDEA* for all density-tightness combinations in the mushy region.

MIDEA* $>$ *CoeEA For 5 density-tightness combinations in the mushy region, both the *MIDEA* and the *CoeEA* failed to solve any of the CSP instances in all their runs. No t-test can be performed on these results. For the other density-tightness combinations, the the *MIDEA* clearly outperformed the the *CoeEA*, as at least the *MIDEA* was able to solve some CSP instances in some of the runs.

The ranking given in equation 8.4 corresponds closely to the one we found in section 8.1 when based on the *SR* measure alone. It differs from the rankings we got in section 8.2 mostly because those were based on the relationships between the *SR-AES* and the *SR-CC*. The ranking given in equation 8.4 however is more accurate than the one given in section 8.1 because through the t-tests it is based on the whole sample of runs and not just on the average of all runs.

8.4 Preliminary Conclusion

The comparison above, as well as the ranking, allows us to give a preliminary conclusion about what we have discovered about evolutionary algorithms for solving constraint satisfaction problems so far. As was to be expected, some algorithms performed consistently better than others. The ranking of the algorithms in the previous section is a reliable indication which algorithms solve more CSP instances in more runs. It does not tell us everything however, for a complete picture the efficiency measures (*AES* and *CC*) have to be considered as well. Common among most algorithms high in the ranking is that they are lower in the ranking when compared in the *SR-AES* and especially in the *SR-CC* plane. This suggests that algorithms which are good at solving CSP instances also need to do a lot of work. In some cases, much of this work is hidden.

Some algorithms performed poorly, notably the *MIDEA* and the *CoeEA*. This in spite of the good performance reported in the papers in which these algorithms were proposed. One reason for this lack of performance could lie in the fact that in this thesis a different CSP test-set was used. We, however, believe, that a good algorithm should perform well on any reasonable test-set of CSP instances, a belief that is supported by the comparable performance of the other algorithms.

The comparison and the ranking also tell us about the effectiveness of the underlying techniques, irrespective of the algorithm which uses it. We found that the co-evolutionary approach, used in the *HPEA* and the *CoeEA*, did not perform well. The co-evolutionary approach necessitates the maintenance of two populations of individuals simultaneously throughout the run. This divides the available amount of fitness evaluations over the two populations and also uses conflict checks for both populations. To offset this investment, the combination of both populations in the co-evolutionary algorithm has to increase performance sufficiently to make it worth while. The co-evolutionary algorithms in the inventory did not show this. Although there is an element of danger of basing conclusions on examples, because of the relatively poor performance of the co-evolutionary algorithms in the inventory, we believe it is safe to conclude that the co-evolutionary approach is not the best technique for solving the constraint satisfaction problem .

Generalising the other techniques used, we believe that all other algorithms in the inventory enhanced the performance of the evolutionary algorithm with the application of some sort of heuristic or local-search technique. From the comparison in Chapter 6 it should be clear why the authors of the algorithms in the inventory have decided to do so. The *IEA* itself does not have enough search power to the problem with a reasonable amount of effort. Although the *IEA* is found to be good at maintaining diversity in the population and thus searches through a large enough portion of the search space, it lacks the depth of search displayed by the *HCAWR* to find solution fast enough. It is only reasonable that the depth-first search of an iterated local-search technique should be combined with the diversity maintaining ability (or breadth-first search) in an evolutionary algorithm as this could improve the performance of the resulting algorithm to supersede both separate algorithms.

A good example of this approach can be found in the three versions of the *HeuristicEA*, where two heuristics were used in two different genetic operators. In the comparison given above, we see that this setup works very well. The heuristics in the genetic operators are used to find good individuals, in effect doing the depth-first search, while the evolutionary mechanism is used to maintain diversity in the population in order to avoid convergence toward a local optimum. In order to get good results however, a delicate balance between the two mechanisms has to be found.

The three versions of the *ArcEA* are also an example of this approach. In these algorithms, progressively more complicated local-search techniques are introduced, unfortunately with progressively less good results. The difference between *ArcEA1* and *HEA1* is small. The different method use for calculating the fitness does not seem to improve the performance however and the performance of the *ArcEA1* seems to be mostly dependent on the asexual heuristic operator from *HEA1*. The exchange of the asexual heuristic operator with the arc genetic operators does not increase the performance, even though in *ArcEA3*, the static arc crossover operator is made dynamic and both arc crossover operators include an intelligent construction method of the individual. We performed a number of parameter adjustment experiments for this algorithm but found no way of improving the performance from the one given, therefore we must conclude that the additions of the *ArcEA* algorithms are not sufficient to ensure better performance. Note, however, that the additions of the *ArcEA* algorithms focus on directing the search on solving constraints that are harder to satisfy while, in our test-set, the tightness of the constraints is approximately equal. On a test-set where there is variance between the tightness constraints in the the CSP instance, the *ArcEA* may well have an edge over the other algorithms in the inventory.

Both the *ESPEA* and the *LSEA* are the most explicit in incorporating a local-search technique. Both algorithms introduce a third operator in the form of a repair operator. There is a drawback in doing this that has to be recognised: because both operators are applied after the genetic operators, there is the possibility of undoing (at least some of) the work of the genetic operators. This is most clear in the *ESPEA*, where a simple repair rule is used to re-label some of the variables in the individual to values that do not conflict with the constraints. In the *LSEA*, although more complicated, the same thing happens because it searches for individuals with a maximum length consistent compound label, removing the other values from the domain sets of the variables. The local-search techniques in both the *ESPEA* and the *LSEA* are very strong, in that the possibility of undoing changes made by the (other) genetic operators is large. Because of this, they can render the genetic operators superfluous, a notion we will investigate further in the next chapter. Of note here is that both local-search techniques used in the *ESPEA* and the *LSEA* can not be tweaked and both use a lot of conflict checks, i.e., hidden work.

The *SAWEA* is different from the other algorithms in that it takes the most direct approach to implementing a local-search technique and uses the evolutionary part of the algorithm only as a way to supply the permutation for decoder. This division of labour has its advantages: the decoder only searches through the viable search-space, discarding domain values that are inconsistent with domain values already labelled. This reduces the search space and makes the algorithm more efficient. However, the *SAWEA* also relegates the evolutionary search process to finding suitable permutations for the decoder and the relation between the fitness value of an individual and the genotype of the individual is less clear as it is obscured by the decoder. Nevertheless, the addition of a local-search technique in the decoder of the *SAWEA* is essential for increasing the performance of the algorithm.

All in all, we found that if one wants to solve constraint satisfaction problems with evolutionary algorithms, the addition of a local-search technique to the algorithm, in order to give it the ability to find good individuals during the run, is important, and from the ranking found in the previous section, the best place to add the local-search technique would be in either the genetic algorithms, as shown by the *HeuristicEA*, or in a third operator that acts as a repair operator, as in the *ESPEA* or the *LSEA*. An outlier so far, but still performing well, is the *SAWEA* which adds a local-search technique in a decoder.

For further study in the thesis we want to reduce the number of algorithms to a more manageable amount, concentrating on the algorithms with the best performance. The algorithms chosen for further study are found through a process of elimination. First and most obvious we eliminate the *MIDEA* and the *CoeEA*. Both algorithms have poor performance in the mushy region of the test-set, *CoeEA* being unable to solve the CSP instance in any of its runs and *MIDEA* unable to solve them in five of the nine density-tightness combinations in the mushy region. Next we eliminate versions of the same algorithm with poorer performance, so for the *HeuristicEA* we only consider *HEA3* and for *ArcEA* we only consider *ArcEA1*. The difference between the *HEA3* and the *ArcEA1* however is small, both share the asexual heuristic operator from the *HeuristicEA*. The performance of the *ArcEA1* is also consistently lower than that of the *HEA3*, so we eliminate *ArcEA1*. The difference between the *SAWEA* and the *HPEA* is not so clear-cut, however, when we look at the rankings based on the *SR-AES* and the *SR-CC* plane, we find that the *SAWEA* has is consistently higher in the ranking than the *HPEA* for both the effectivity-efficiency plane comparisons, so we eliminate *HPEA* as well. For the rest of the thesis we therefore consider only the following four algorithms (in order of the ranking given in equation 8.4):

1. the *LSEA*;
2. the *HEA3*;
3. the *ESPEA*; and
4. the *SAWEA*.

Chapter 9

De-Evolutionarising Evolutionary Algorithms, Memetic Overkill, and the Superior Evolutionary Algorithm

This chapter describes the notion of *de-evolutionarising* evolutionary algorithms to find out if they are susceptible to what we term *memetic overkill*. Of the four best performing evolutionary algorithms in the inventory, only *SAWEA* is found not to suffer from memetic overkill. This algorithm is then adjusted to construct the superior evolutionary algorithm for solving the constraint satisfaction problem by introducing four variants. None of the variants was found to suffer from memetic overkill. The best performing variant is selected as the superior performing evolutionary algorithm.

9.1 De-evolutionarising Evolutionary Algorithms

In Chapter 8 we found that the four algorithms with the best performance all include a heuristic or a local-search technique. The power of the heuristic and local-search technique and the way they are used, both influence the amount of improvement of the performance. Here, we investigate the influence of the evolutionary components of these algorithms on their performance. This is done by removing the evolution from the algorithms, a process we term *de-evolutionarising* the algorithm. The influence of the evolutionary component is determined by comparing the performance of the original algorithm with the de-evolutionarised variant. Technically speaking the question is how to de-evolutionarise the algorithms. To answer this question, we consider the essential features of the evolutionary algorithm for which it holds that after removing these features, the algorithm would not be evolutionary. There are three essential features that make an algorithm evolutionary:

1. a population of candidate solutions;
2. variation operators (e.g., crossover and mutation); and
3. natural selection (i.e., selection based on the fitness of an individual).

Although all three features are closely related, the first two are in part dependent on each other, because without a population of candidate solutions, the crossover operator can not be used. Furthermore, examples exist of evolutionary algorithms without these features. In evolutionary strategies ([7, 80]) examples exist that do not maintain a population of candidate solutions. These examples have a population of only one individual. Evolutionary programming ([39, 37]) does not have crossover operators, or any other form of recombination, although they use a mutation operator.

Taking these considerations into account, we de-evolutionarise evolutionary algorithms by removing first natural selection and second the population (by setting the population size to one). When an evolutionary algorithm includes a crossover operator, this is removed together with the population.

As for natural selection, recall that there are two selection steps in the general evolutionary algorithm framework: parent selection and survivor selection. For either of them we say that it represents natural selection if a fitness-based bias is incorporated, favouring better candidates. Note, that an evolutionary algorithm does not need to have natural selection in both steps. For instance, generational genetic algorithms use only parent selection (and all children survive), while evolutionary strategies use only survivor selection (and parents are selected uniform randomly). However, an evolutionary algorithm must have fitness-bias in at least one of these steps. If neither parent selection nor survivor selection are performed by using fitness-bias (e.g., by uniform random selection) then no natural selection is done and random walk is obtained.

Considering the role of the population, the common evolutionary computation wisdom states that population size of one is a singularity, i.e., it is a special case of the general scheme, for ‘real’ evolution more individuals are needed.

In practice we *de-evolutionarise* the evolutionary algorithms in two steps and create two new variants for each algorithm. In the first variant we use uniform random selection for both parent and survivor selection, thereby switching off natural selection. In the second variant we switch off natural selection *and* use a population size of one (and consequently cease to use crossover when necessary). In the following overview we denote these variants as *EA*, *EA-sel*, *EA-sel-pop*.

Based on the observations in the previous chapter we de-evolutionarise only the best performing algorithms in the inventory. In order of the ranking given in the previous

$(p_1, \overline{p_2})$	***LSEA*** *SR*	*AES*	*CC*	***LSEA*-sel** *SR*	*AES*	*CC*	***LSEA*-sel-pop** *SR*	*AES*	*CC*
$(0.1, 0.9)$	1.0	13	8893	1.0	13	8893	1.0	9	4387
$(0.2, 0.9)$	0.988	540	300212	0.988	540	300212	1.0	154	87068
$(0.3, 0.8)$	0.812	9825	4714303	0.812	9825	4714303	1.0	1058	568152
$(0.4, 0.7)$	0.808	5935	2641589	0.808	5935	2641589	1.0	1024	533308
$(0.5, 0.6)$	0.924	10124	4307145	0.924	10124	4307145	1.0	910	461629
$(0.6, 0.6)$	0.752	12080	4573046	0.752	12080	4573046	1.0	1360	781702
$(0.7, 0.5)$	0.776	11562	4673916	0.776	11562	4673916	1.0	1618	861174
$(0.8, 0.5)$	0.796	11422	4279512	0.796	11422	4279512	1.0	1377	794020
$(0.9, 0.4)$	0.936	4097	1689760	0.936	4097	1689760	1.0	738	381452

Table 9.1: Comparison of the *LSEA*, *LSEA*-sel, and *LSEA*-sel-pop.

chapter, the following algorithms were de-evolutionarised: the *LSEA*, the *HEA3*, the *ESPEA*, and the *SAWEA*. The results of the experiments are shown in Tables 9.1, 9.2, 9.3, and 9.4. We experimented only on the density-tightness combinations in the mushy region of the test-set and the tables include the *SR*, *AES*, and *CC* measures. The first column indicates the density-tightness combinations for which the results are given. The results in the second to fourth column of each table are copied from the inventory. The fifth to seventh column show the results of the first variant of each algorithm (EA-sel) and the eighth to tenth column show the results for the second variant of each algorithm (EA-sel-pop).

Table 9.1 shows no difference between the *SR*, the *AES*, and the *CC* values of the original *LSEA* and the *LSEA*-sel. This suggests that natural selection is completely overruled by the repair operator in the *LSEA*. The table also shows that the performance of the *LSEA*-sel-pop is better than both the original *LSEA* and *LSEA*-sel. The *LSEA*-sel-pop solves all CSP instances in all runs for all density-tightness combinations in the mushy region of the test-set and does so using (on average) fewer evaluations and fewer conflict checks. The decrease of *AES* and *CC* is significant, sometimes as much as nearly one tenth of the evaluations or conflict checks are used. From the results it is clear that the repair operator of the *LSEA* on its own is powerful enough to solve the CSP instances in the test-set and that natural selection and the use of a population (and a crossover operator) actually decrease the performance of the algorithm. As such, the ability of the *LSEA* to solve the CSP comes from the local-search technique used while the evolutionary components of natural selection and the use of a population are actually harmful to the performance of the algorithm.

Table 9.2 shows that the performance of the *HEA3*-sel is better than the performance of the original *HEA3*. For some density-tightness combinations in the mushy region of the test-set (e.g., $(0.6, 0.6)$) the the *SR* is more than doubled (going from 0.44 to 0.956). This shows that natural selection is actually harmful for the performance of the *HEA3* and that the local-search techniques in the heuristic operators are powerful enough to find solutions to the CSP instance in almost all runs. The differ-

$(p_1, \overline{p_2})$	**_HEA3_** *SR*	*AES*	*CC*	**_HEA3_-sel** *SR*	*AES*	*CC*	**_HEA3_-sel-pop** *SR*	*AES*	*CC*
$(0.1, 0.9)$	1.0	26	23899	1.0	27	25138	1.0	7	5364
$(0.2, 0.9)$	0.984	419	621391	1.0	221	320560	1.0	62	51241
$(0.3, 0.8)$	0.688	1635	2489261	1.0	952	1435814	1.0	185	156040
$(0.4, 0.7)$	0.712	1404	2110238	1.0	404	603560	0.988	140	118541
$(0.5, 0.6)$	0.692	2382	3647367	0.996	717	1083481	0.956	99	83752
$(0.6, 0.6)$	0.44	988	1493377	0.956	1618	2467666	0.948	220	187711
$(0.7, 0.5)$	0.588	969	1472759	0.988	1960	2982026	0.972	202	172835
$(0.8, 0.5)$	0.488	1258	1932541	0.976	3601	5538668	0.972	211	182419
$(0.9, 0.4)$	0.76	1563	2404978	1.0	912	1393405	0.972	121	104850

Table 9.2: Comparison of the *HEA3*, *HEA3*-sel, and *HEA3*-sel-pop.

ences between the *AES* and the *CC* measures of the two variants is more varied. Although the *AES* of the *HEA3*-sel is less in density-tightness combinations $(0.2, 0.9)$, $(0.3, 0.8)$, $(0.4, 0.7)$, $(0.5, 0.6)$, and $(0.9, 0.4)$, it is increased for density-tightness combinations $(0.1, 0.9)$, $(0.6, 0.6)$, $(0.7, 0.5)$, and $(0.8, 0.5)$. For the *CC* measure, in density-tightness combinations $(0.1, 0.9)$, $(0.2, 0.9)$, $(0.3, 0.8)$, $(0.4, 0.7)$, $(0.5, 0.6)$, and $(0.9, 0.4)$ the *HEA3*-sel used fewer conflict checks while for density-tightness combinations $(0.6, 0.6)$, $(0.7, 0.5)$, and $(0.8, 0.5)$ is increased. Although the performance of the *HEA3*-sel-pop is slightly lower than that of the *HEA3*-sel, it is still much better than that of the original *HEA3*. The reason for the slight decrease is probably the removal of the heuristic multi-parent crossover operator when the population of the *HEA3*-sel-pop was set to one. Still, the performance of the *HEA3*-sel-pop is better that that of the original *HEA3*, so also in this case, we conclude that the local-search technique used in the remaining heuristic operator is powerful enough to solve CSP instances on its own. Therefore, although the use of a population through the heuristic multi-parent operator is still useful, natural selection decreases the performance of the algorithm.

Table 9.3 shows a dramatic improvement of the performance of the *ESPEA*-sel over the original *ESPEA*. Without natural selection, the *ESPEA* is able to solve all CSP instance in the mushy region of the test-set in all runs. Apart from density-tightness combination $(0.1, 0.9)$ the efficiency measured by the *AES* and *CC* also shows an improvement. There is no more improvement in *SR* between the *ESPEA*-sel and the *ESPEA*-sel-pop, but since all CSP instances in the mushy region of the test-set are solved by both the *ESPEA*-sel and the *ESPEA*-sel-pop, this is not possible. However, the *ESPEA*-sel-pop improved the efficiency of the algorithm even further, probably because no evaluations and conflict checks are used to maintain the population. Overall, the increase in performance in the *ESPEA*-sel and *ESPEA*-sel-pop variants is dramatic, which suggests that the local-search technique used in the repair operator of the *ESPEA* is powerful enough to solve the CSP on its own. The use of the evolutionary components of natural selection and the use of a population are harmful to the performance of the *ESPEA*.

$(p_1, \overline{p_2})$	***ESPEA*** *SR*	***AES***	***CC***	***ESPEA*-sel** *SR*	***AES***	***CC***	***ESPEA*-sel-pop** *SR*	***AES***	***CC***
$(0.1, 0.9)$	1.0	45	14920	1.0	48	17598	1.0	18	6231
$(0.2, 0.9)$	0.952	2404	924530	1.0	275	179191	1.0	137	80635
$(0.3, 0.8)$	0.728	6165	2670936	1.0	629	423241	1.0	222	132072
$(0.4, 0.7)$	0.844	6021	2785182	1.0	529	346895	1.0	170	99313
$(0.5, 0.6)$	0.844	4839	2415945	1.0	442	297149	1.0	170	96408
$(0.6, 0.6)$	0.8	6015	3039882	1.0	736	492493	1.0	238	152962
$(0.7, 0.5)$	0.772	9241	4738977	1.0	839	557504	1.0	275	162950
$(0.8, 0.5)$	0.84	9241	2497913	1.0	1218	788666	1.0	236	155603
$(0.9, 0.4)$	0.944	3589	2085063	1.0	374	272451	1.0	161	96214

Table 9.3: Comparison of the *ESPEA*, *ESPEA*-sel, and *ESPEA*-sel-pop.

$(p_1, \overline{p_2})$	***SAWEA*** *SR*	***AES***	***CC***	***SAWEA*-sel** *SR*	***AES***	***CC***	***SAWEA*-sel-pop** *SR*	***AES***	***CC***
$(0.1, 0.9)$	0.92	56	13679	0.0	*undef.*	*undef.*	0.28	693	115256
$(0.2, 0.9)$	0.72	849	173181	1.0	165	27925	0.08	18709	3441713
$(0.3, 0.8)$	0.6	2134	412326	0.257	27946	5343048	0.08	16704	3012739
$(0.4, 0.7)$	0.664	5975	1111019	0.422	11239	1955642	0.296	17066	2950401
$(0.5, 0.6)$	0.772	9511	1731587	0.633	12422	2153396	0.26	18497	3169435
$(0.6, 0.6)$	0.64	3326	603370	0.368	7820	1371509	0.192	24009	4152730
$(0.7, 0.5)$	0.32	6481	1170229	0.071	30848	5450022	0.14	13126	2289924
$(0.8, 0.5)$	0.396	2393	438562	0.284	5239	899806	0.204	19084	3274588
$(0.9, 0.4)$	0.828	3547	645733	0.633	21519	3823496	0.304	10159	1809621

Table 9.4: Comparison of the *SAWEA*, *SAWEA*-sel, and *SAWEA*-sel-pop.

In contrast to the first three algorithms, Table 9.4 shows that the performance of both the *SAWEA*-sel and the *SAWEA*-sel-pop decreases when natural selection and the use of a population is removed. Both evolutionary components benefit the performance of the *SAWEA*. This is especially clear for density-tightness combination $(0.1, 0.9)$ where the original *SAWEA* solved the CSP instance in almost all runs while for both the *SAWEA*-sel and the *SAWEA*-sel-pop none (for *SAWEA*-sel) or few (for *SAWEA*-sel-pop) were solved. Comparing the original *SAWEA* and the *SAWEA*-sel, only for density-tightness combination $(0.2, 0.9)$ was there an improvement in the *SR*, the *AES*, and the *CC*. There is no clear reason for this improvement and we see it as a random occurrence. Overall, however, the performance decreases from the original *SAWEA* to the *SAWEA*-sel, and again to the *SAWEA*-sel-pop, and we conclude that natural selection and the use of a population is beneficial to the performance of the *SAWEA* and that the power to solve the CSP comes not only from the local-search technique used in the decoder but also from the evolutionary components of the algorithm.

9.2 Memetic Overkill

In section 8.4 we concluded that the best way to improve the performance of an evolutionary algorithm is to incorporate a heuristic or local-search mechanism. In the previous section however, we showed that for the best four algorithms in the inventory, three of them increased performance when we de-evolutionarised them. Obviously great care has to be taken when incorporating a heuristic or local-search technique in an evolutionary algorithm because when the heuristic or local-search technique is too strong the evolutionary components of the algorithm may actually reduce performance.

The best examples of this are the *LSEA* and the *ESPEA*. Both algorithms incorporate powerful local-search techniques in a third (repair) operator. The results shown in the previous section show that the local-search techniques on their own are powerful enough to solve the CSP instances in the test-set and that, in fact, the evolutionary components of natural selection and the use of a population decreases the performance of the algorithm.

The *HEA3* differs from the *LSEA* and the *ESPEA* in that the heuristics are incorporated in the variation operators of the algorithm itself. The heuristics themselves are well-known and commonly used but as in the *LSEA* and the *ESPEA*, when natural selection was removed from the algorithm, the performance of the algorithm increased. When in addition the use of a population was removed from the algorithm, and consequently the use of the (multi-parent) crossover operator as well, the performance of the algorithm decreased somewhat but was still superior to the original algorithm. As with the *LSEA* and the *ESPEA*, the evolutionary components of natural selection, and to a lesser extend the use of a population decreases the performance of the *HEA3*.

Only the *SAWEA* showed a decrease in performance when natural selection and the use of a population is removed from the algorithm. This leads to the conclusion that in the *SAWEA*, these evolutionary components still have a positive effect on the performance of the algorithm.

The effect of the evolutionary components having a negative effect on the performance of the algorithm we call *memetic overkill*. The term is derived from the term used to describe evolutionary algorithms incorporating heuristic or local-search techniques: memetic algorithms. As said before, the incorporation of heuristic or local-search techniques in evolutionary algorithms in order to improve their performance is common place. However, when the incorporated techniques are too powerful, their incorporation in an evolutionary algorithm can actually hamper the performance of these techniques, resulting in memetic overkill.

Although the consequences of memetic overkill and the ways of testing whether it occurs are explained above, the reason for it to occur is not. We believe that there are two interrelated reasons for memetic overkill to occur: the way in which the local-search techniques are used, and the power of the local-search technique itself.

In the best examples of memetic overkill, the *LSEA* and the *ESPEA*, the local-search technique is incorporated in a third (repair) operator. This operator is applied after the variation operators of the algorithms and is therefore allowed to over-rule the (quite) random choices of these variation operators. As such, there is a chance that the repair operator will undo some of the changes that the variation operators have made. Because the local-search technique makes its choices (in part) deterministic, their application after the variation operators makes the search less random, in effect making the search less diverse. In this respect, the local-search techniques provide a more depth-first search while the evolutionary components of natural selection and the use of a population provide a more breadth-first search. In the *LSEA* and the *ESPEA*, the constant struggle of the local-search techniques to do a depth-first search (in order to find a solution fast) with the evolutionary components to do a breadth-first search (in order to maintain diversity) leads to a lower performance of the algorithm as a whole. When the breadth-first search of the evolutionary components is removed, therefore, the performance is improved.

This is closely related to the power of the local-search technique, for if the local-search technique is not powerful enough to find the solution of the problem on its own, the breadth-first search of the evolutionary components allow the algorithm more avenues for the local-search technique to solve the problem. This should increase the overall performance of the algorithm. The power of the local-search technique on its own, independent of the way it is incorporated in the algorithm, can be enough to lead to memetic overkill. The *HEA3* is a clear example of this. In the *HEA3*, the heuristics are incorporated in the variation operators of the algorithm, so the way in which the techniques are incorporated does not pose a problem. The heuristics themselves, however, are so capable of finding a solution, that the evolutionary components attempts to do a breadth-first search (that is, to maintain diversity) reduces the performance of the algorithm. We believe that the randomising effect of the evolutionary components harms the performance because of the different avenues the algorithms investigates ultimately either do not lead to a solution of the problem, or use up so many of the available search steps that the algorithm is terminated before it can find a solution.

So, how to reconcile the incorporation of a heuristic or local-search technique with memetic overkill? Apparently, the heuristic or local-search technique must be placed in such a way that it can not undo too many (random) changes of the variation operators, and, it must not be overly powerful in its guidance toward solving the problem (in this case, the CSP). In short, the focus that the depth-first search of a heuristic or local-search technique provides must be balanced with the diversity or breadth-first search that the evolutionary components provide.

An algorithm wherein this balance has been achieved is the *SAWEA*. Although the *SAWEA* does not have as good a performance as the *LSEA*, the *HEA3*, and the *ESPEA*, it does not suffer from memetic overkill. We believe that the reason for this is that the *SAWEA* consists of two parts: the local-search decoder and the evolutionary permutation searcher to supply the decoder. Although the performance of the *SAWEA* depends on both parts of the algorithm, they are independent in that the local-search technique in the decoder is not directly incorporated in the evolutionary part of the algorithm. Also, the local-search technique used in the decoder is not powerful enough to solve the CSP on its own. The two parts of the *SAWEA* algorithm are connected through the stepwise adaptation of weights fitness function, which focusses the evolutionary part of

		Evolution	
		With	*Without*
Heuristics	*Weak*	Good	Poor
	Strong	Inferiour	Good

Table 9.5: Performance of algorithms that incorporate weak, strong, or no heuristics and evolution.

the *SAWEA* towards finding better permutations for the decoder through the candidate solutions that the decoder provides. The result of this is that the local-search technique used in the decoder is balanced against the evolutionary part of the algorithm, neither has the upper hand and both can work together to achieve a higher performance.

We can generalise the relative performance of algorithms based on whether they incorporate either weak or strong heuristics and evolution or not. Table 9.5 shows the four possible combinations and they relative performance. Unsurprisingly, algorithms that incorporate weak heuristics and no evolution have a poor performance. The de-evolutionarised variants of the *LSEA*, the *HEA3*, and the *ESPEA* show that when an algorithm incorporates a strong heuristic but no evolution the performance is (or rather, can be) good. When an algorithm combines strong heuristics with evolution however, the performance is inferior to the algorithm which does not incorporate evolution. The *SAWEA* on the other hand showed that an algorithm incorporating weak heuristic and evolution can still have good performance.

A strange situation can arise when one wants to increase the performance of an evolutionary algorithm by incorporating either more and more local-search techniques or incorporating more and more powerful local-search techniques into the algorithm. There is a point in this process where incorporating more, or more powerful local-search techniques actually makes the evolutionary components of the algorithm have a negative effect on the performance. At this stage one is better off continuing without the evolutionary components, i.e., using the algorithm as a pure iterated local-search algorithm instead of an evolutionary algorithm. Because in this design process one starts of with a simple evolutionary algorithm and progressively embellishes it with local-search techniques, the effect described above is also known as the *stone soup* effect (see also [68]). It is historically ironic to find out that when researchers started to incorporate more, or more powerful heuristics in their evolutionary algorithms as way of boosting their performance, they would have been, in the end, better off without the evolutionary components of their evolutionary algorithms.

9.3 Adjustments to make the Superior EA

Since the *LSEA*, the *ESPEA*, and the *HEA3* all suffer from memetic overkill, further tweaking of these algorithms in order to improve their performance as evolutionary algorithms seems pointless. Although the *SAWEA* had the poorest performance of the four algorithms tested, it still is the best candidate to adjust in order to construct a superior performance evolutionary algorithm, the main goal of this thesis. There are several ways of doing this. The most obvious method is to increase the power of the local-search technique in the decoder. However, increasing the power of the local-search technique, for example by incorporating a backtracking algorithm, makes the *SAWEA* vulnerable to memetic overkill, so this is not a viable option. We already tried to increase the performance of the *SAWEA* by making adjustments to the evolutionary part of the algorithm in [17] without much success. Now, we opt for focussing on using information gained during the run of the algorithm to improve the performance. We hope that this increases the performance of the algorithm without increasing the risk of memetic overkill.

In order to describe how we want to improve the performance of the *SAWEA*, we have to describe in more detail how the greedy local-search technique of the decoder works. The decoder in the *SAWEA* takes a permutation evolved by the evolutionary part of the algorithm and uses a greedy algorithm to convert this into a, possibly partial, solution of the CSP instance to solve. This is done by iteratively labelling a variable in the permutation, in order, with a value from its domain. The value is taken from the domain set of that variable. In the original *SAWEA*, the domain set is ordered by the value of the domain value in ascending order. For example, the test-set used in this thesis includes CSP instances with a uniform domain size of 10, the the domain set used by the *SAWEA* is: $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. As a result, the first time a variable in the permutation is labelled by the decoder, it is labelled with the value 1.

The greedy algorithm in the decoder itself is clearly not powerful enough to solve a complex CSP instance, i.e., a CSP instance in the mushy region. When the greedy algorithm has to label a variable for which all domain values in the domain set violates a constraint relevant to an already labelled variable, it leaves it unlabelled. The number of unlabelled variables of a decoded individual is then used as the basis for the fitness value of that individual.

The variants of the *SAWEA* recognise that the ordering of the elements of the domain sets is chosen quite arbitrarily. Prior knowledge about how to order the elements of the domain sets, however, is easy to obtain, although this will cost a certain number of conflict checks. This cost, however, will be incurred only once, at the initialisation of the algorithm. The idea is to use the restrictiveness of a value to order the domain set of a variable. This is calculated by counting the number of constraint violations when that value is checked against all other values of all other variables. This is analogous to counting the number of times that a certain label is in the set of compound labels of all constraints of a CSP instance. By excluding double counting, the number of conflict checks needed can be decreased. If the label is in more constraints it is more restrictive than if it is not.

We investigate two domain set orderings: one where the values are ordered in ascending restrictiveness; and one where the values are ordered in descending restrictiveness. The idea behind the first ordering is that values which are less restricted are better candidates for labelling that variable. The idea behind the second ordering is that values which are more restricted should be used earlier in the search. One could say that the

$(p_1, \overline{p_2})$	***SAWEA r1*** ***SR***	***AES***	***CC***	***SAWEA r1*-sel** ***SR***	***AES***	***CC***	***SAWEA r1*-sel-pop** ***SR***	***AES***	***CC***
$(0.1, 0.9)$	0.948	654	104901	1.0	836	132676	1.0	851	140990
$(0.2, 0.9)$	0.956	3716	743856	0.996	10559	1949775	0.744	24149	4408818
$(0.3, 0.8)$	0.92	7201	1359534	0.936	13750	2461121	0.6	24282	4341539
$(0.4, 0.7)$	1.0	4861	872589	0.92	8650	1508664	0.648	22563	3972954
$(0.5, 0.6)$	1.0	5945	1058857	1.0	7859	1363549	0.82	20587	3590547
$(0.6, 0.6)$	1.0	6474	1156792	0.996	8972	1554420	0.708	25492	4457152
$(0.7, 0.5)$	1.0	7325	1302119	0.988	10185	1778085	0.684	27640	4835898
$(0.8, 0.5)$	1.0	5882	1039437	1.0	11068	1924934	0.72	24612	4297115
$(0.9, 0.4)$	1.0	4292	761993	1.0	4471	788815	0.932	17540	3115641

Table 9.6: Comparison of the *SAWEA r1*, *SAWEA r1*-sel, and *SAWEA r1*-sel-pop.

first ordering is an easiest-first ordering while the second ordering is a hardest-first ordering. Apart from the original ordering of the domain sets, we also included a test ordering, in which the domain sets were ordered randomly. In total four variants will be considered:

1. ascending domain set ordering by value;
2. random domain set ordering;
3. ascending domain set ordering based on restrictiveness; and
4. descending domain set ordering based on restrictiveness.

Note that the first two orderings are problem independent while the last two orderings are problem dependent.

We added another alteration to the original *SAWEA*. This involves intermittently re-ordering the values in the domain sets during the run of the algorithm. At intervals equal to the update interval for the weights of the stepwise adaptation of weights mechanism, the domain sets of the variables that remained unlabelled in the individual with the best fitness value are rotated. Rotating a domain sets means that the first value (element) in the domain set replaces the last value in the domain set and that all other values in the domain set replace the one preceding it. In essence, the first domain value in the domain set becomes the last, the second the first, and so on. Other re-orderings of the domain sets were tried as well but the naive rotating of domain sets had the best results. Rotating domain sets explicitly uses information gained during the run of the algorithm, namely which variables so far have been difficult to label using the current domain sets ordering. The idea is that by using this information, the performance of the algorithm will be improved.

Combining the rotation method with the four domain set orderings we get four variants:

SAWEA r1 dynamically rotates domain sets ordered in ascending order by value;

$(p_1, \overline{p_2})$	***SAWEA r2*** SR	AES	CC	***SAWEA r2*-sel** SR	AES	CC	***SAWEA r2*-sel-pop** SR	AES	CC
$(0.1, 0.9)$	1.0	64	9665	1.0	103	16432	1.0	294	48093
$(0.2, 0.9)$	0.988	1750	350789	0.992	5646	1043625	0.752	23471	4338731
$(0.3, 0.8)$	0.956	3986	763903	0.952	9801	1761384	0.624	25697	4623096
$(0.4, 0.7)$	0.976	3598	652045	0.972	5088	897387	0.688	20651	3639388
$(0.5, 0.6)$	1.0	3166	557026	1.0	3859	669803	0.868	19695	3396530
$(0.6, 0.6)$	1.0	4024	715122	0.992	5298	921481	0.732	21208	3661156
$(0.7, 0.5)$	1.0	4878	864249	1.0	7153	1249932	0.7	20746	3610806
$(0.8, 0.5)$	1.0	5762	1012082	1.0	7139	1240297	0.712	21344	3701840
$(0.9, 0.4)$	1.0	2333	408016	1.0	2609	461836	0.94	15529	2741701

Table 9.7: Comparison of the *SAWEA r2*, *SAWEA r2*-sel, and *SAWEA r2*-sel-pop.

$(p_1, \overline{p_2})$	***SAWEA r3*** SR	AES	CC	***SAWEA r3*-sel** SR	AES	CC	***SAWEA r3*-sel-pop** SR	AES	CC
$(0.1, 0.9)$	1.0	106	22026	1.0	233	42641	1.0	644	113599
$(0.2, 0.9)$	1.0	2263	462630	0.996	6020	1124397	0.76	25278	4664481
$(0.3, 0.8)$	0.992	5476	1045548	0.992	8890	1603512	0.62	31127	5560629
$(0.4, 0.7)$	0.96	5208	948532	0.96	6163	1094412	0.752	21072	3727604
$(0.5, 0.6)$	1.0	3549	630359	0.988	5283	924150	0.824	21214	3686307
$(0.6, 0.6)$	1.0	5727	1007768	0.996	6546	1142049	0.692	22902	3998333
$(0.7, 0.5)$	1.0	8155	1450130	0.996	7732	1355025	0.66	24453	4274086
$(0.8, 0.5)$	1.0	6090	1062279	0.996	8364	1453261	0.724	21930	3832569
$(0.9, 0.4)$	1.0	2833	504622	1.0	2333	416026	0.888	16717	2960015

Table 9.8: Comparison of the *SAWEA r3*, *SAWEA r3*-sel, and *SAWEA r3*-sel-pop.

$(p_1, \overline{p_2})$	***SAWEA r4*** SR	AES	CC	***SAWEA r4*-sel** SR	AES	CC	***SAWEA r4*-sel-pop** SR	AES	CC
$(0.1, 0.9)$	1.0	52	12193	1.0	87	18209	1.0	191	35730
$(0.2, 0.9)$	0.964	1925	389564	0.996	5597	1046514	0.708	21787	4034458
$(0.3, 0.8)$	1.0	3495	674248	0.992	7360	1336169	0.652	25496	4558643
$(0.4, 0.7)$	0.96	4169	758786	0.956	5157	910049	0.704	23412	4098368
$(0.5, 0.6)$	1.0	2944	523872	1.0	3369	586291	0.868	19864	3462682
$(0.6, 0.6)$	1.0	2951	531129	0.992	5661	990433	0.712	22056	3853155
$(0.7, 0.5)$	1.0	4424	789253	1.0	5281	927072	0.736	21837	3810733
$(0.8, 0.5)$	1.0	5434	962742	1.0	6319	1102868	0.772	22875	3966539
$(0.9, 0.4)$	1.0	2324	416441	1.0	1780	319268	0.92	13545	2398367

Table 9.9: Comparison of the *SAWEA r4*, *SAWEA r4*-sel, and *SAWEA r4*-sel-pop.

	$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
SAWEA r1 $<$ *SAWEA r2*	-	0.01	1.0	0.0	-	-	-	-	-
	-	0.99	0.5	1.0	-	-	-	-	-
	$=$	$<$	$=$	$<$	$=$	$=$	$=$	$=$	$=$
SAWEA r1 $<$ *SAWEA r3*	-	0.0	0.01	0.06	-	-	-	-	-
	-	1.0	0.99	0.97	-	-	-	-	-
	$=$	$<$	$<$	$<$	$=$	$=$	$=$	$=$	$=$
SAWEA r1 $<$ *SAWEA r4*	-	0.38	0.0	0.06	-	-	-	-	-
	-	0.81	1.0	0.97	-	-	-	-	-
	$=$	$<$	$<$	$<$	$=$	$=$	$=$	$=$	$=$
SAWEA r2 $>$ *SAWEA r3*	-	0.08	0.01	0.31	-	-	-	-	-
	-	0.096	0.99	0.16	-	-	-	-	-
	$=$	$>$	$<$	$>$	$=$	$=$	$=$	$=$	$=$
SAWEA r2 $>$ *SAWEA r4*	-	0.08	0.0	0.31	-	-	-	-	-
	-	0.04	1.0	0.16	-	-	-	-	-
	$=$	$>$	$<$	$>$	$=$	$=$	$=$	$=$	$=$
SAWEA r3 $=$ *SAWEA r4*	-	0.0	0.16	1.0	-	-	-	-	-
	-	0.0	0.92	0.5	-	-	-	-	-
	$=$	$>$	$<$	$=$	$=$	$=$	$=$	$=$	$=$

Table 9.10: t-test results for the ranking *SAWEA r1*, *SAWEA r2*, *SAWEA r3*, and *SAWEA r4* on *SR*.

SAWEA r2 dynamically rotates domain sets ordered randomly;

SAWEA r3 dynamically rotates domain sets ordered in ascending restrictiveness; and

SAWEA r4 dynamically rotates domain sets ordered in descending restrictiveness.

We used the same test-set as used before for our experiments on these four variants. We also de-evolutionarised each variant, introducing two de-evolutionarised variants for each variant, one where natural selection is removed, and one where both natural selection and the population are removed. As before, we term these variants *-sel* and *-sel-pop*. The results of these experiments are shown in Tables 9.6, 9.7, 9.8, and 9.9.

Tables 9.6, 9.7, 9.8, and 9.9 show that all four variants of the *SAWEA* have higher *SR* than the original *SAWEA*. The biggest improvement was seen for density-tightness combinations $(0.7, 0.5)$ and $(0.8, 0.5)$ where the *SR* went from 0.32 and 0.396 respectively to 1.0 for all four variants. The efficiency of the four variants however was lower than the original *SAWEA*, both the *AES* and the *CC* are higher. The big increase in *SR* however outweighs the relatively small increase of the *AES* and *CC*.

To answer the question of which variant performed best we return to a statistical analysis of the results through t-tests. Because the *SR* results of the experiments are so close together we analyse the *AES* and *CC* results as well as the *SR* results of the experiments. Table 9.10 shows the analysis of the *SR* results, Table 9.11 the analysis of

	$(\mathbf{0.1}, \mathbf{0.9})$	$(\mathbf{0.2}, \mathbf{0.9})$	$(\mathbf{0.3}, \mathbf{0.8})$	$(\mathbf{0.4}, \mathbf{0.7})$	$(\mathbf{0.5}, \mathbf{0.6})$	$(\mathbf{0.6}, \mathbf{0.6})$	$(\mathbf{0.7}, \mathbf{0.5})$	$(\mathbf{0.8}, \mathbf{0.5})$	$(\mathbf{0.9}, \mathbf{0.4})$
SAWEA r1 $>$ *SAWEA r2*	0.0	0.0	0.0	0.05	0.0	0.0	0.0	0.86	0.0
	0.0	0.0	0.0	0.02	0.0	0.0	0.0	0.43	0.0
	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$=$	$>$
SAWEA r1 $\gtrsim$ *SAWEA r3*	0.0	0.0	0.1	0.46	0.0	0.4	0.27	0.75	0.0
	0.0	0.0	0.05	0.77	0.0	0.2	0.86	0.63	0.0
	$>$	$>$	$>$	$\lesssim$	$>$	$>$	$<$	$=$	$>$
SAWEA r1 $>$ *SAWEA r4*	0.0	0.0	0.0	0.39	0.0	0.0	0.0	0.47	0.0
	0.0	0.0	0.0	0.19	0.0	0.0	0.0	0.24	0.0
	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$\gtrsim$	$>$
SAWEA r2 $<$ *SAWEA r3*	0.02	0.14	0.04	0.03	0.39	0.03	0.0	0.67	0.09
	0.99	0.93	0.98	0.99	0.81	0.98	1.0	0.66	0.96
	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$\lesssim$	$<$
SAWEA r2 $\gtrsim$ *SAWEA r4*	0.41	0.70	0.51	0.32	0.56	0.01	0.37	0.67	0.98
	0.2	0.65	0.26	0.84	0.28	0.01	0.19	0.33	0.49
	$>$	$\lesssim$	$\gtrsim$	$<$	$\gtrsim$	$>$	$>$	$\gtrsim$	$\gtrsim$
SAWEA r3 $>$ *SAWEA r4*	0.0	0.3	0.01	0.17	0.15	0.0	0.0	0.37	0.11
	0.0	0.15	0.0	0.08	0.07	0.0	0.0	0.19	0.05
	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$

Table 9.11: t-test results for the ranking *SAWEA r1*, *SAWEA r2*, *SAWEA r3*, and *SAWEA r4* on *AES*.

the *AES* results, and Table 9.12 the analysis of the *CC* results. Based on this analysis a ranking for each of the three measures can be given. The *SR* measure, in this respect, has to be maximised, while the *AES* and *CC* measures have to be minimised.

The ranking for the *SAWEA* variants based on the *SR* measure is shown in equation 9.1. In Table 9.10 however, it is seen that for 6 out of the 9 density-tightness combinations in the mushy region, the *SR* results of the four variants are equal. For these 6 density-tightness combinations all four variants solve all CSP instances in all runs. Therefore, the difference upon which the *SR* ranking is based is calculated over 3 density-tightness combinations only. Overall, *SAWEA r2* showed the best *SR* of all four variants while *SAWEA r3* and *SAWEA r4* had about equal *SR*, *SAWEA r1* had the lowest *SR* of all four variants.

$$\textit{SAWEA r2} \quad > \quad \textit{SAWEA r3} \quad = \quad \textit{SAWEA r4} \quad > \quad \textit{SAWEA r1} \qquad (9.1)$$

Table 9.11 shows that the *AES* results of the four variants had more variance over all density-tightness combinations in the mushy region. The ranking of the four variants based on the *AES* measure is found in equation 9.2. As before, the best performing algorithm is shown to the left of the ranking but as the *AES* (as the *CC*) measure is to be minimised the comparative signs between the algorithms are reversed. The *SAWEA r4*

	$(\mathbf{0.1},$ $\mathbf{0.9})$	$(\mathbf{0.2},$ $\mathbf{0.9})$	$(\mathbf{0.3},$ $\mathbf{0.8})$	$(\mathbf{0.4},$ $\mathbf{0.7})$	$(\mathbf{0.5},$ $\mathbf{0.6})$	$(\mathbf{0.6},$ $\mathbf{0.6})$	$(\mathbf{0.7},$ $\mathbf{0.5})$	$(\mathbf{0.8},$ $\mathbf{0.5})$	$(\mathbf{0.9},$ $\mathbf{0.4})$
SAWEA r1 $>$ *SAWEA r2*	0.0 0.0 $>$	0.0 0.0 $>$	0.0 0.0 $>$	0.06 0.03 $>$	0.0 0.0 $>$	0.0 0.0 $>$	0.0 0.0 $>$	0.82 0.41 $\lesssim$	0.0 0.0 $>$
SAWEA r1 $\gtrsim$ *SAWEA r3*	0.0 0.0 $>$	0.0 0.0 $>$	0.12 0.06 $>$	0.4 0.8 $>$	0.0 0.0 $>$	0.34 0.17 $>$	0.27 0.87 $<$	0.84 0.58 $\lesssim$	0.0 0.0 $>$
SAWEA r1 $>$ *SAWEA r4*	0.0 0.0 $>$	0.0 0.0 $>$	0.0 0.0 $>$	0.45 0.23 $\gtrsim$	0.0 0.0 $>$	0.0 0.0 $>$	0.0 0.0 $>$	0.49 0.24 $\gtrsim$	0.0 0.0 $>$
SAWEA r2 $<$ *SAWEA r3*	0.0 1.0 $<$	0.11 0.94 $<$	0.04 0.98 $<$	0.03 0.99 $<$	0.35 0.82 $<$	0.04 0.98 $<$	0.0 1.0 $<$	0.71 0.64 $\lesssim$	0.06 0.97 $<$
SAWEA r2 $\lesssim$ *SAWEA r4*	0.21 0.9 $<$	0.67 0.66 $\lesssim$	0.56 0.28 $\gtrsim$	0.31 0.85 $<$	0.63 0.32 $\gtrsim$	0.02 0.01 $<$	0.41 0.21 $\lesssim$	0.72 0.36 $\gtrsim$	0.88 0.56 $\gtrsim$
SAWEA r3 $>$ *SAWEA r4*	0.0 0.0 $>$	0.29 0.14 $>$	0.01 0.01 $>$	0.16 0.08 $>$	0.15 0.08 $>$	0.0 0.0 $>$	0.0 0.0 $>$	0.44 0.22 $\gtrsim$	0.12 0.06 $>$

Table 9.12: t-test results for the ranking *SAWEA r1*, *SAWEA r2*, *SAWEA r3*, and *SAWEA r4* on *CC*.

algorithm used less than or similar amounts of *AES* than the *SAWEA r2* algorithm. The *SAWEA r2* algorithm was more efficient than the *SAWEA r3* algorithm which in turn used less than or similar amounts of *AES* than the *SAWEA r1* algorithm.

$$\textit{SAWEA r4} \quad \lesssim \quad \textit{SAWEA r2} \quad < \quad \textit{SAWEA r3} \quad \lesssim \quad \textit{SAWEA r1} \qquad (9.2)$$

The ranking based on the *CC* measure is shown in equation 9.3. Based on the analysis shown in Table 9.12, the ranking is very similar to the *AES* ranking shown in equation 9.2 except for the *CC* measure the *SAWEA r2* and *SAWEA r4* algorithms are reversed.

$$\textit{SAWEA r2} \quad \lesssim \quad \textit{SAWEA r4} \quad < \quad \textit{SAWEA r3} \quad \lesssim \quad \textit{SAWEA r1} \qquad (9.3)$$

Based on the statistical analysis we can conclude that the *SAWEA r2* is the best performing variant of *SAWEA*. Although it was ranked second on the *AES* measure, it was ranked first on the *CC* measure and more importantly, first on the *SR* measure. The fact that *SAWEA r1* was ranked last on all three measures demonstrates that the original domain sets ordering (in ascending order by value) is not the best ordering to use and that the decision to reorder the elements of the domain sets resulted in an

$(\mathbf{p_1}, \overline{\mathbf{p_2}})$	*LSEA*	*ESPEA*	*HEA3*	*SAWEA r2*
(0.1,0.9)	1.0	1.0	1.0	1.0
(0.2,0.9)	0.988	0.984	0.952	0.988
(0.3,0.8)	0.812	0.688	0.728	0.956
(0.4,0.7)	0.808	0.712	0.844	0.976
(0.5,0.6)	0.924	0.692	0.844	1.0
(0.6,0.6)	0.752	0.44	0.8	1.0
(0.7,0.5)	0.776	0.588	0.772	1.0
(0.8,0.5)	0.796	0.488	0.84	1.0
(0.9,0.4)	0.936	0.76	0.944	1.0

Table 9.13: Comparison of the *SR* of the *LSEA*, *ESPEA*, *HEA3*, and the *SAWEA r2*.

increased performance. Comparing the ordering based on the restrictiveness of a value in the domain set of a variable (in *SAWEA r3* and *SAWEA r4*) the orderings show that ordering the domain set in descending restrictiveness increased the performance more the ordering the domain set in ascending restrictiveness. It appears that re-labelling hardest-first outperforms easiest-first. In general, however, ordering the domain sets randomly outperformed all other variants. Although surprising, this domain set ordering is bias-free and does not use conflict checks to come to an ordering (as do the orderings in *SAWEA r3* and *SAWEA r4*) and we recommend this ordering for further use.

The *SAWEA r2* is then the superior evolutionary algorithm. Comparing the *SR* of the *LSEA*, *ESPEA*, and the *HEA3* and the *SAWEA r2* in Table 9.13 shows that the *SAWEA r2* has a superior performance when these algorithms are not de-evolutionarised. Also, the *SAWEA r2* does not suffer from memetic overkill, which the other three algorithm do suffer from. A further boon is that the *SAWEA r2* is a variant that does not need problem dependent information to achieve its good performance.

Chapter 10

Conclusions

The main motivation for writing this thesis is our belief that for many problems evolutionary computation can provide a viable alternative to other algorithms. In this thesis we test if this also holds for the constraint satisfaction problem. The test we use is to construct a superior evolutionary algorithm and compare its performance to alternative methods for solving the constraint satisfaction problem.

An evolutionary algorithm is not the most obvious method to solve the constraint satisfaction problem since it does not contain a built-in objective function to optimise. Because of the many applications based on the problem however, the problem has received a lot of attention from the evolutionary computation community. A large number of evolutionary algorithms for solving the constraint satisfaction problem have been proposed in the last two decades.

Comparing the performance of these algorithms based on literature was hampered because of the different test-sets used, some of which were found to be deficient in some respects. Additionally, different ways to measure the performance of the algorithms were used further obscuring the relative performance of the algorithms.

In this thesis we offer a solution to these problems by the construction of a new test-set using the latest random constraint satisfaction problem generator and explicitly defining the measures on which the performance of the evolutionary algorithms are compared. A representative subset of the algorithms proposed in literature was re-implemented in a uniform manner using a basic experimentation platform thus making a fair comparison possible.

The relative performance of the algorithms was compared based on the defined measures, statistical analysis of the measurements and different performance measures were compared relative to each other as well. Further experimentation on the four best performing algorithms revealed that three of them suffered from memetic overkill. Memetic overkill occurs when an evolutionary algorithm incorporating a strong heuristic or local-search technique has inferior performance to the algorithm without the evolutionary components. As three out of the four best performing algorithms suffer from memetic overkill, constructing the superior evolutionary algorithm by combining the effective components from these algorithms is of no use, since it would only result in a new algorithm suffering from memetic overkill.

Instead the superior evolutionary algorithm was constructed from the one algorithm not suffering from memetic overkill. Because the incorporation of more or more powerful heuristics would probably lead to this algorithm also suffering from memetic overkill, the decision was made to instead use information gained during the run to enhance the performance of the algorithm. Earlier investigation of the algorithm has already shown that modifications to the evolutionary components do not increase the performance of the algorithm.

From the four proposed variants of the algorithm, one was found to have superior performance. The algorithm uses randomly ordered domain elements and rotation to label variables in the decoder part of the algorithm. The algorithm is called *SAWEA r2* and was found not to suffer from memetic overkill and have superior performance to the evolutionary algorithms previously investigated.

What remains is to compare the performance of this algorithm with alternative methods to solving the constraint satisfaction problem to see if our above mentioned belief is justified.

10.1 Evolutionary and Classical Algorithms

The performance of the *SAWEA r2* is compared to the *Hill Climber with Restart Algorithm* (*HCAWR*) from Chapter 5, and the *Chronological Backtracking Algorithm* (*CBA*), and the *Forward Checking with Conflict-Directed Backjumping Algorithm* (*FCCDBA*) from Chapter 3. The *HCAWR* is an iterated local-search algorithm while both the *CBA* and the *FCCDBA* are classical algorithms. The *CBA* and the *FCCDBA* are both complete algorithms and because the constructed test-set from Chapter 4 includes only solvable instances, the *SR* performance measure will always be 1.0 for these algorithms. Also note that because the *CBA* and the *FCCDBA* are deterministic algorithms, only one run for each CSP instance in the test-set is necessary, additional runs will show the same results. The *AES* performance measure, although in some measure applicable to the *HCAWR*, is not applicable to the classical algorithms. This leaves only the *CC* measure to compare the performance of the four algorithms.

Table 10.1 shows the results from the experiments with the *SAWEA r2*, the *HCAWR*, the *CBA*, and the *FCCDBA* on the mushy region of the test-set. Only the *SAWEA r2* has an *SR* of less than 1.0 for density-tightness combinations $(0.2, 0.9)$, $(0.3, 0.8)$, and $(0.4, 0.7)$, all other algorithms, and for the *CBA* and the *FCCDBA* we knew this, solve all the CSP instances in all their runs. The *SR* of the *SAWEA r2* however is very close to 1.0, only 3, 11, and 6 runs out of a total of 250 were unsuccessful for density-tightness combinations $(0.2, 0.9)$, $(0.3, 0.8)$, and $(0.4, 0.7)$ respectively.

For the *CC* performance measure we find that the *SAWEA r2* is more efficient than the *HCAWR* but less than the *FCCDBA*. For density-tightness combination $(0.1, 0.9)$, the

$(p_1, \overline{p_2})$	***SAWEA r2*** *SR*	***SAWEA r2*** *CC*	***HCAWR*** *SR*	***HCAWR*** *CC*	***CBA*** *SR*	***CBA*** *CC*	***FCCDBA*** *SR*	***FCCDBA*** *CC*
$(0.1, 0.9)$	1.0	9665	1.0	234242	1.0	3800605	1.0	930
$(0.2, 0.9)$	0.988	350789	1.0	1267015	1.0	335166	1.0	3913
$(0.3, 0.8)$	0.956	763903	1.0	2087947	1.0	33117	1.0	2186
$(0.4, 0.7)$	0.976	652045	1.0	2260634	1.0	42559	1.0	4772
$(0.5, 0.6)$	1.0	557026	1.0	2237419	1.0	23625	1.0	3503
$(0.6, 0.6)$	1.0	715122	1.0	2741567	1.0	44615	1.0	5287
$(0.7, 0.5)$	1.0	864249	1.0	3640630	1.0	35607	1.0	4822
$(0.8, 0.5)$	1.0	1012082	1.0	2722763	1.0	28895	1.0	5121
$(0.9, 0.4)$	1.0	408016	1.0	2465975	1.0	15248	1.0	3439

Table 10.1: Comparison of the *SAWEA r2*, the *HCAWR*, the *CBA*, and the *FCCDBA*.

SAWEA r2 is more efficient than the *CBA*, but for the other density-tightness combinations this is reversed. Note here that the *SR* of the *SAWEA r2* can be increased by increasing the maximum number of evaluations allowed or alternatively by running the *SAWEA r2* multiple times. Given the disparity between the *CC* of the *SAWEA r2* and the *HCAWR*, the *SAWEA r2* could be applied several times before the number *CC* of the *HCAWR* would be exhausted. However, the difference between the *CC* of the *SAWEA r2* and the classical algorithms significant, the *FCCDBA* in particular being more efficient by a large margin.

So are evolutionary algorithms a viable alternative to other algorithms for solving the CSP? Yes, and no. The *SAWEA r2* does have almost the same *SR* as classical algorithms, and by allowing longer runs, we believe that it can attain an *SR* of 1.0 for all density-tightness combinations in the mushy region of the test-set. However, although the *SAWEA r2* is more efficient than the *HCAWR*, it is far less efficient than the *FCCDBA*. Of note here is that were the *SAWEA r2* is the best performing algorithm of its class, the *HCAWR* is probably not. Better (read more efficient) iterated local-search algorithms do exist. The conclusion therefore must be that if getting a solution fast (efficient), the *SAWEA r2*, and in general an evolutionary algorithm is *not* a viable alternative.

So far in the thesis we have concentrated our comparison of methods to solve the CSP purely on performance. Within a scientific context this makes sense. However, from the standpoint of a user, other factors besides performance might be of importance. In that context, evolutionary algorithms have two things in their favour: general applicability and ease of design.

Although all evolutionary algorithms in this thesis were specifically designed to solve the CSP, they are usually also applicable to other related problems. The *SAWEA*, for example, has been used to solve the satisfiability problem and the graph colouring problem and has shown good performance there. It has also been shown to be useful in solving data mining problems, much less related to the constraint satisfaction problem. The classical algorithms in this thesis however are less applicable to solve other problems than the ones for which they were designed, although the basic techniques used in them might still be useful.

In general, evolutionary algorithms are also easy to invent and design. The *SAWEA*, although more difficult than an off-the-shelf evolutionary algorithm like the *IEA*, is still relatively easy to design. Although evolutionary algorithms have a fair amount of parameters to fine-tune, some guidelines for setting these parameters are available, while overall, the evolutionary paradigm used in the algorithms is quite robust for all but the most outlandish parameter settings. In the end, evolution has the tendency to find a solution to a problem eventually, as can be observed in nature. And although the *CBA* is also easy to design (and implement), the length of the pseudo-code for the *FCCDBA* (given in Chapter 3) clearly indicates that it is not. The increase in efficiency of the *FCCDBA* then comes from more research a-priori into solving the problem. For the user unwilling to invest in this, evolutionary algorithms are an alternative with the additional benefit that they can be applied to a wider variety of problems.

Thus, for the user interested primarily in finding a solution to a problem and unwilling to invest much effort in trying to understand the intricacies of it, evolutionary algorithms *are* a viable alternative. The *SAWEA r2* then is an illustration that evolutionary algorithms are up to this task.

10.2 Main Contributions of the Thesis

In the course of the investigation presented in this thesis, the following main contributions to the scientific community were made:

- a methodology for constructing a test-set of CSP instances, tailored especially for comparing the performance of iterated local-search algorithms, evolutionary algorithms in particular;
- a comprehensive inventory of eight evolutionary algorithms for solving the constraint satisfaction problem including full descriptions of the algorithms and experimental results for accessing their performance.
- a methodology for comparing and ranking the performance of evolutionary algorithms using traditional and statistical methods, and comparison of the relative performance in the effectivity-efficiency plane;
- offering the notion of memetic overkill and a methodology for identifying if an algorithm suffers from memetic overkill by de-evolutionarising it;
- a platform for experimental research into evolutionary algorithms for solving the constraint satisfaction problem including a uniform implementation of a comprehensive inventory of evolutionary algorithms; and
- a well-founded conclusion on a superior performing evolutionary algorithm for solving the randomly generated binary constraint satisfaction problem.

10.3 Future Research

Although we hope that the contributions made in this thesis, because of the solid experimental basis on which they are founded, will be useful for researchers, they also pose a number of new avenues for future research.

Memetic overkill is probably not only a problem for evolutionary algorithms solving the constraint satisfaction problem. It has to be expected that it occurs for evolutionary algorithms solving other problems as well. Further research into the extent of memetic overkill happening in evolutionary algorithms for other problems might therefore provide interesting results.

No research was done on the performance of the evolutionary algorithms when the size of the CSP instances was increased. These scale-up experiments will provide valuable insight in how, for example, the *SAWEA r2* can handle an increase in problem size. Classical algorithms encounter a performance barrier with the increase of combinatorial complexity. It is possible that evolutionary algorithms are less affected by this and that they will outperform classical algorithms in scale-up experiments.

And finally, the constraint satisfaction problems solved by the algorithms were 'artificial', in that they were all generated by a random CSP generator. For scientific research this works best, but in real-life, problems often contain structures that make them different from randomly generated ones. Although the *SAWEA r2* has good performance on randomly generated CSP instances, comparing its performance on real-life problems might provide insight in how the algorithm can handle these kinds of problems.

Bibliography

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, July 1990.

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

[3] D. Applegate, W. Cook, and A. Rohe. Chained lin-kernigham for large traveling salesman problems. Technical Report 99887, Forschungsinstitute für Diskrete Mathematik, University of Bonn, Germany, 1999.

[4] W. Atmar. The inevitability of evolutionary invention. Unpublished Manuscript, 1979.

[5] W. Atmar. Notes on the simulation of evolution. *IEEE Transactions on Neural Networks*, 5(1):130–147, 1994.

[6] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th International Conference on Artificial Intelligence – ICAI98*, pages 311–318, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[7] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, NY, 1996.

[8] T. Bäck, D. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*, New York, 1997. Institute of Physics Publishing Ltd, Bristol and Oxford University Press.

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

[10] E.B. Baum. Iterated descent: A better algorithm for local search in combinatorial optimisation problems. Manuscript, Caltech, Pasadena, CA, 1986.

[11] E.B. Baum. Toward practical "neural" computation for combinatorial optimisation problems. In J. Denker, editor, *Neural Networks for Computing*, AIP Conference Proceedings, pages 53–64, 1986.

[12] J. Baxter. Local optima avoidance in depot location. *Journal of the Operation Research Society*, 32:815–819, 1981.

[13] L. Booker. Improving search in genetic algorithms. In *Genetic Algorithms and Simulated Annealing*, pages 61–73. Morgan Kaufmann Publisher, Inc., 1987.

[14] J. Bowen and G. Dozier. Solving constraint satisfaction problems using a genetic/systematic search hybrid that realizes when to quit. In L.J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms – ICGA95*, pages 122–129. Morgan Kaufmann Publishers, Inc., 1995.

[15] P. Cheeseman, B. Kenefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings on the International Joint Conference on Artificial Intelligence – IJCAI91*, pages 331–337, 1991.

[16] S.A. Cook. The complexity of theorem-proving procedures. In *The complexity of theorem-proving procedures*, pages 151–158, Shaker Heights, Ohio, 1971.

[17] B.G.W. Craenen and A.E. Eiben. Stepwise adaptation of weights with refinement and decay on constraint satisfaction problems. In L. Spector, E. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO2001*, pages 291–298, San Francisco, CA, 2001. Morgan Kaufmann, Inc.

[18] B.G.W. Craenen, A.E. Eiben, and E. Marchiori. Solving constraint satisfaction problems with heuristic-based evolutionary algorithms. In *Proceedings of the Congress on Evolutionary Computation 2000 – CEC2000*, pages 1571–1577. IEEE Computer Society Press, July 2000.

[19] B.G.W. Craenen, A.E. Eiben, E. Marchiori, and A. Steenbeek. Combining local search and fitness function adaptation in a genetic algorithm for solving binary constraint satisfaction problems. In D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO2000*, page 381. Morgan Kaufmann Publishers, Inc., 2000.

[20] J.M. Crawford and L.D. Anton. Experimental results on the crossover point in satisfiability problems. In R. Fikes and W. Lehnert, editors, *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 21–27, Menlo Park, California, 1993. AAAI Press.

[21] C. Darwin. *The Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London, 1859.

[22] R. Dechter. On the expressiveness of networks with hidden variables. In T. Dietterich and W. Swartout, editors, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 556–562, Hynes Convention Centre, 1990. MIT Press.

[23] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[24] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based micro-genetic algorithm. In ICEC94 [50], pages 306–311.

[25] 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 – ICEC95*, pages 614–619. IEEE Computer Society Press, 1995.

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

[27] A.E. Eiben. Evolutionary algorithms and constraint satisfaction: Definition, survey, methodology, and research directions. In L. Kallel, B. Naudts, and A. Rogers, editors, *Theoretical Aspects of Evolutionary Computing*, Natural Computing Series, pages 13–58. Springer, 2001.

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

[29] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In ICEC94 [50], pages 542–547.

[30] 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.

[31] A.E. Eiben and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In ICEC96 [51], pages 258–261.

[32] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003. ISBN 3-540-40184-9.

[33] A.E. Eiben and J.K. van der Hauw. Adaptive penalities for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution '97 – AE97*, volume 1363 of *Lecture Notes on Computer Science*, pages 95–106. Springer-Verlag, Berlin, 1998.

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

[35] A.E. Eiben and J.I. van Hemert. SAW-ing EAs: Adapting the fitness function for solving constrained problems. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 389–402. McGraw-Hill, 1999.

[36] T.C. Fogarty. Varying the probability of mutation in the genetic algorithm. In Schaffer [78], pages 104–109.

[37] D.B. Fogel. *Evolutionary Computation*. IEEE Computer Society Press, 1995.

[38] D.B. Fogel. *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1st edition, 1998.

[39] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, 1966.

[40] A. Fukunaga. Restart scheduling for genetic algorithms. In A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature – PPSN98*, volume 1498 of *Lecture Notes in Computer Science*, pages 357–366. Springer-Verlag, Berlin, 1998.

[41] S. Golomb and L. Baumert. Backtrack programming. *A.C.M.*, 12(4):516–524, 1965.

[42] J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem. *Journal of Evolutionary Computation*, 10(1):35–50, 2002.

[43] J.J. Grefenstette. Optimisation of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):122–128, 1986.

[44] H. Handa, N. Baba, O. Katai, T. Sawaragi, and T. Horiuchi. Genetic algorithm involving coevolution mechanism to search for effective genetic information. In ICEC97 [52], pages 709–714.

[45] H. Handa, C.O. Katai, N. Baba, and T. Sawaragi. Solving constraint satisfaction problems by using coevolutionary genetic algorithms. In *Proceedings of the 5th IEEE Conference on Evolutionary Computation – ICEC98*, pages 21–26. IEEE Computer Society Press, 1998.

[46] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint-satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[47] A. Hoffman. *Arguments on Evolution: A Paleontologist's Perspective*. Oxford University Press, New York, 1988.

[48] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Harbor, 1975.

[49] J. Huxley. The evolutionary process. In J. Huxley, A.C. Hardey, and E.B. Ford, editors, *Evolution as a Process*, pages 9–33. Collier Books, New York, 1963.

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

[51] *Proceedings of the 3rd IEEE Conference on Evolutionary Computation – ICEC96*. IEEE Computer Society Press, 1996.

[52] *Proceedings of the 4th Conference on Evolutionary Computation – ICEC97*. IEEE Society Press, 1997.

[53] D.S. Johnson. Local optimisation and the traveling salesman problem. In *Proceedings of the 17th Colloquium on Automata, Languages, and Programming*, volume 443 of *LNCS*, pages 446–461, Berlin, 1990. Springer Verlag.

[54] D.S. Johnson and L.A. McGeoch. The travelling salesman problem: A case study in local optimization. In E.H.L. Aarst and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, Chichester, England, 1997.

[55] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89(1–2):365–387, 1989.

[56] E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. Random constraint satisfaction: theory meets practice. In M. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming – CP98*, pages 325–339. Springer Verlag, 1998.

[57] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In Bäck [9], pages 330–337.

[58] E. Marchiori. Genetic, iterated and multistart local search for the maximum clique problem. In *Applications of Evolutionary Computing*, volume 2279 of *LNCS*, pages 112–121. Springer, 2002.

[59] E. Marchiori and A. Steenbeek. A genetic local search algorithm for random binary constraint satisfaction problem. In *Proceedings of the 14th Annual Symposium on Applied Computing*, pages 463–469, 2000.

[60] O. Martin and S.W. Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63:57–75, 1996.

[61] O. Martin, S.W. Otto, and E.W. Felten. Large-step markov chains for the traveling salesman problem. *Complex Systems*, 5(3):299–251, 1991.

[62] E. Mayr. *Toward a New Philosophy of Biology: Observation of an Evolutionist*. Belknap Press, Cambridge, 1988.

[63] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

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

[65] D. Mitchell, B. Selman, and H. Levesque. Hard and easy districutions of SAT problems. In *Proceedings of the 10th International Conference on Artificial Intelligence – ICAI-92*, pages 459–465, San Jose, CA, 1992.

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

[67] H. Mühlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In R. Männer and B. Manderick, editors, *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature — PPSN92*, pages 15–25. Elsevier Science Press, 1992.

[68] B. Paechter, T. Fogarty, E. Burke, A. Cumming, and B. Ranking. Stone soup. In Edmund K. Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III — Third International Conference, PATAT 2000*, volume 2079 of *Lecture Notes in Computer Science*, pages 103–106, Konstanz, Germany, 2001. Springer-Verlag.

[69] E.M. Palmer. *Graphical Evolution. An introduction to the theory of random graphs.* Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons, Ltd., Chichester, 1985.

[70] J. Paredis. Coevolutionary constraint satisfaction. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature – PPSN94*, volume 886 of *Lecture Notes in Computer Science*, pages 46–55. Springer Verlag, 1994.

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

[72] J. Paredis. Coevolving cellular automata: Be aware of the red queen. In Bäck [9], pages 393–400.

[73] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[74] M.-C. Riff Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves csp. In ICEC96 [51], pages 279–284.

[75] M.-C. Riff Rojas. Evolutionary search guided by the constraint network to solve csp. In ICEC97 [52], pages 337–348.

[76] M.-C. Riff Rojas. A network-based adaptive evolutionary algorithm for constraint satisfaction problems. *Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 325–339, 1998.

[77] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constrain satisfaction problems. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, pages 550–556, Stockholm, 1990. Pitman.

[78] J.D. Schaffer, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms*, San Mateo, California, 1989. Morgan Kaufmann Publisher, Inc.

[79] J.D. Schaffer, R.A. Caruana, L.J. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In Schaffer [78], pages 51–61.

[80] H.-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, New York, NY, 1995.

[81] 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. Wiley, 1994.

[82] B.M. Smith and M.E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(12):155–181, 1996.

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

[84] J.K. van der Hauw. Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's thesis, Leiden University, 1996.

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

[86] L. van Valen. A new evolutionary law. *Evolutionary Theory*, (1):1–30, 1973.

[87] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Schaffer [78], pages 116–123.

[88] D.E. Wooldridge. *The Mechanical Man: The Physical Basis of Intelligent Life*. McGraw-Hill, New York, 1968.

Index