Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark

# Three EA Techniques for Constrained Parameter Optimization Problems

Master's Thesis
by
Jørgen Bang Erichsen

*To my family*

# Abstract

Three EA techniques for constrained parameter optimization problems are investigated in this thesis. The techniques are

- A genetic algorithm with a fitness function incorporating a penalty weight.
- An ant system inspired by ants foraging for food.
- A genetic algorithm in combination with a decoder function.

All three techniques are explained in detail and their performance on a number of benchmark problems are reported. Furthermore, the thesis contains an introduction to genetic algorithms and constrained optimization problems.

# Preface

In the recent years Evolutionary Algorithms (EA) techniques have been increasingly popular for solving real-life optimization problems like scheduling problems. Almost all real-life problems are constrained in some way or another. For the scheduling problem this might be hard constraints, like a teacher not being able to teach two classes at the same time, and soft constraints like a nurse who prefers to have every second Friday off. In this thesis three EA techniques for constrained parameter optimization problems are presented and compared.

The thesis is structured as follows: The first chapter is an introduction to genetic algorithms, constrained parameter optimization problems and the benchmark problems. In the second chapter the penalty weight approach is presented. The next chapter contains information about a system inspired by ants foraging for food which can be used to find solutions to constrained parameter optimization problems. The decoder approach is presented in chapter four and the last chapter contains a comparison of the three techniques and a concluding discussion.

I would like to thank Zbigniew Michalewicz for spawning my interest in constrained parameter optimization problems and for some inspiring conversations. Thanks also goes to Brian Mayoh for being my advisor and for his ability to always find some interesting articles in the area of Artificial Intelligence. Furthermore, I would like to thank Kim Burgaard who had the tedious task of proof reading this entire thesis. Last but not least I would like to thank my family, Rikke and Tina for always being supportive and by just being there.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This chapter introduces genetic algorithms, constrained optimization problems and the test cases which are the benchmark problems used to compare the performance of the three techniques presented in this thesis.

## 1.1 Genetic algorithms

### 1.1.1 The algorithm

The basic genetic algorithm is presented in figure 1.1. The flow of the program is the following:

1. The first thing to do, is to initialize the population of individuals.
2. Before entering the main loop the initial population needs to be evaluated.
3. A cycle in the main loop begins with a selection of “parents”. This sub-population is used to create the offspring that eventually becomes the new population.
4. Offspring are created from the parents using recombination.
5. With a small probability, the offspring are subject to mutation.
6. The final steps of the cycle is to evaluate the offspring and replace the existing population by the new one.

In the next few sections various aspects of the genetic algorithm will be discussed briefly.

```
procedure genetic algorithm
begin
    g := 0
    Initialize(Population)
    Evaluate(Population)
    while (StopCriterionNotSatisfied) do
    begin
        g := g + 1
        Parents = SelectParents(Population)
        Offspring = Recombine(Parents)
        Mutate(Offspring)
        Evaluate(Offspring)
        Population = Offspring
    end
end
```

Figure 1.1: GA pseudo code

### 1.1.2 Terminology

The following explains figure 1.2 and some of the terminology used in the next sections. The *population* consists of a number of individuals. Each *individual* is a chromosome and a corresponding fitness value. Each *chromosome* is built of genes. A *gene* can be one of several types discussed in the next section.

### 1.1.3 Representation

The first thing to consider, when one wants to use genetic algorithms to find solutions to a certain problem, is the representation. In other words, we have to find a way to encode the solutions to the problem such that the solutions can be expressed by the chromosomes.

Depending on the type of problem, some types of representation are more natural than others. The most commonly used representations are:

**Binary representation** Each chromosome is either 0 or 1. This was the preferred way to represent problems in the beginning of the era of genetic algorithms. Furthermore, much of the theoretical work concerning genetic algorithms has been done on the basis of binary representation.

**Real-valued representation** As the name suggests, the single chromosome is a

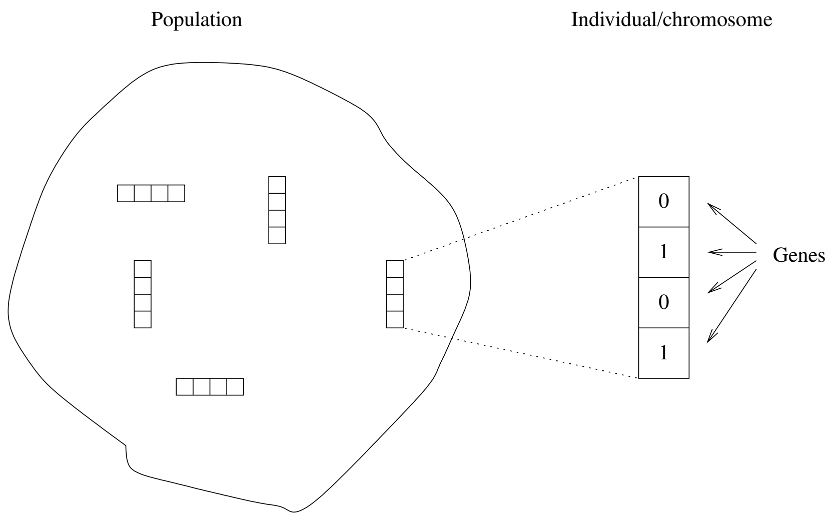

Figure 1.2: The population and an individual/chromosome

real value. This representation is the most commonly used and also used in all the genetic algorithms in this thesis.

**Other representations** For example integer representation and a representation that resembles trees. When finding solutions to the Traveling Salesman Problem (TSP), the integer representation is often used. Lisp expressions can be conceived as syntactic trees and in the field of *genetic programming* the tree representation is used.

### 1.1.4 Evaluation

Each chromosome is assigned a fitness the following way: The chromosome (genotype) is *decoded* to obtain an individual (phenotype). The *fitness function* is then used to calculate the fitness of the individual (phenotype) and this value is assigned to the chromosome (genotype) as the fitness.

Consider the following example. We have a chromosome $c$ with 4 binary genes, 1010. The chromosome is then decoded to obtain an integer, $x = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 10$. The fitness is calculated using the (trivial) fitness function $F(x) = x - 3$ yielding $F(10) = 7$. The chromosome $c$ is hence assigned a fitness value of 7.

In most genetic algorithms the process of decoding the genotype to obtain a phenotype is rather trivial, but it can be arbitrarily complex. One example of a non-trivial decoder is used in one of the three techniques presented in this thesis.

### 1.1.5 Selection

Selection is the process of selecting individuals from the existing population suitable for creating new offspring. Several methods of selection exist and some of the more commonly used are:

**Roulette wheel selection** Each individual is selected with a probability that is the individual's fitness divided by the average fitness of the whole population. This is done by "spinning a roulette wheel" where each individual has a slot with size proportional to its fitness compared to the sum of all individual's fitness.

**Rank selection** Each individual is assigned a rank using the fitness of the individual compared to the rest of the population. This means that the fittest individual will have rank 1 and the individual with the worst fitness will have the lowest rank. Using the rank, the individuals can for example be selected with probability $p(r) = q(1-q)^{r-1}$ where $r$ is the rank and $q$ is a user-defined parameter.

**Tournament selection** A number of random individuals are selected from the population and the fittest is used. This is repeated until the desired number of individuals have been selected. When the number of a random individuals from the population is two, the selection scheme is referred to as *binary tournament selection.*

Three issues concerning selection remain to be addressed. The first one is the concept of *elitism.* When elitism is used, the fittest individual from the old population is copied with no modifications to the new population. The number of individuals copied may be higher than one, but is usually one.

Another remaining issue is the concept of *steady state genetic algorithms*. Traditional genetic algorithms are generational in the sense that the new population replaces the old generation entirely. In steady state genetic algorithms on the other hand, only a few new individuals are created and they replace the same number of individuals in the old generation. When using traditional genetic algorithms and elitism, the generations overlap to a small degree but with steady state genetic algorithms this generational overlap is much more pronounced, although not due to elitism.

Finally, the concept of *premature convergence* is strongly related to selection. Sometimes a genetic algorithm converges to a suboptimal local optimum due to lack of variance in the population. This phenomenon is termed premature convergence. Especially genetic algorithms using roulette wheel selection are prone to premature convergence. If one individual is much fitter than the rest, it will be selected very often for breeding using the roulette wheel selection. This super

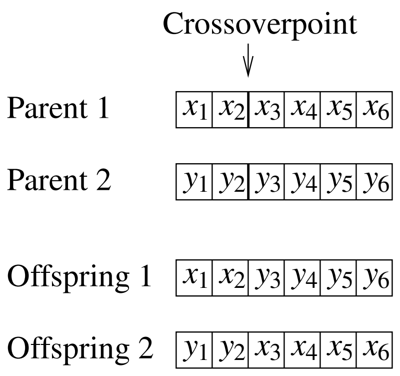
Crossoverpoint
Parent 1
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$
Parent 2
$y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$
Offspring 1
$x_1$ $x_2$ $y_3$ $y_4$ $y_5$ $y_6$
Offspring 2
$y_1$ $y_2$ $x_3$ $x_4$ $x_5$ $x_6$

Figure 1.3: Example of one-point crossover

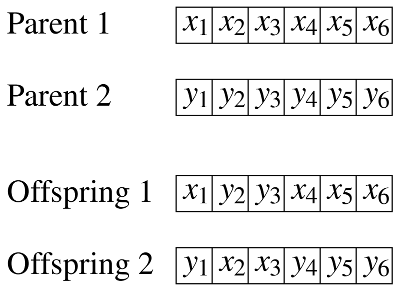
Parent 1
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$
Parent 2
$y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$
Offspring 1
$x_1$ $y_2$ $y_3$ $x_4$ $x_5$ $x_6$
Offspring 2
$y_1$ $x_2$ $x_3$ $y_4$ $y_5$ $y_6$

Figure 1.4: Example of uniform crossover

individual will quickly dominate the population and the variety will be very low. This problem is not very pronounced with rank selection and binary tournament selection.

### 1.1.6 Genetic operators

Two types of genetic operators are presented: Crossover and mutation. The operators are applied with probabilities $p_c$ and $p_m$ respectively.

#### Recombination

The notion of recombination refers to the creation of new individuals using the genetic information from existing individuals. This is done using crossover. Crossover is in general considered the most important genetic operator in genetic algorithms and is the feature which distinguishes genetic algorithms from other evolutionary algorithms. The operator comes in several varieties.

**One-point crossover** Two parents are selected and one point in the chromosome is selected as the crossover point. The first offspring is created by taking the genes from Parent 1 from the beginning to the crossover point and the remaining genes from Parent 2. The second offspring is created in an equal manner but with the roles of the parents reversed. The situation is depicted in figure 1.3.

**Two-point crossover** Two-point crossover is very similar to one-point crossover, the only difference being that there now are two crossover-points. The first offspring is created by taking the first genes from the beginning of the chromosome to the first crossover point from parent 1, the next genes from the first crossover point to the second are taken from parent 2, and the remaining genes from parent 1. The two-point crossover can in a similar fashion be extended to the more general $n$-point crossover.

**Uniform crossover** The uniform crossover works in the following way. For each gene, the gene comes from parent 1 if a random number $r$ with $0 \leq r \leq 1$ is larger than $p_x$ and from parent 2 otherwise. The value of $p_x$ is often but not always set to 0.5. An example of two parents and two offspring created by uniform crossover is depicted in figure 1.4.

**Arithmetic crossover** The arithmetic crossover is used in genetic algorithms where the genotype is represented using floating point representation. Two parents create one offspring by taking the mean of the their genes. In other words if we have parent 1 with genes $x_1, \ldots x_n$ and parent 2 with genes $y_1, \ldots y_n$ the offspring will have genes of the form $(x_1 + y_1)/2, \ldots, (x_n + y_n)/2$.

### Mutation

Mutation is the random change of one or more genes of each chromosome. Each gene is changed with probability $p_m$. With a binary representation this corresponds to flipping the bit, ie from 0 to 1 or from 1 to 0. With real-valued representation, a number of new possibilities arise.

**Simple mutation** With simple mutation each gene selected for mutation is *replaced* by a random value from the domain of the gene.

**Biased mutation** With biased mutation, a random value is *added* to the old value of the gene if the gene is selected for mutation. The random value added, is often produced using a Gaussian distribution with mean 0 and a standard deviation of 10% of the problem domain. This kind of mutation is often referred to as *Gaussian mutation*. With biased mutation it is possible to go

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.1 | 0.2 | **0.8** | 0.4 | 0.5 | 0.6 | 0.9 |

Original chromosome

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.1 | 0.2 | **0.2** | 0.4 | 0.5 | 0.6 | 0.9 |

Simple mutation

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.1 | 0.2 | **0.9** | 0.4 | 0.5 | 0.6 | 0.9 |

Biased mutation

Figure 1.5: Example of simple mutation and biased mutation

beyond the bounds of a gene and this has to be handled in some way. For example, a gene could have values in the domain $[0,1]$ and have an actual value of 0.95. If the gene is mutated and a value of 0.1 is added, the new value will be 1.05 which is not legal. One way to fix this, is to just set the value of the gene to the boundary value — 1.0 in the example — whenever this happens.

Examples of the two types of mutations are given in figure 1.5. The third gene with the value 0.8 has been selected for mutation. In the case of simple mutation the value 0.8 is replaced by the value 0.2. In the case of biased mutation, the value 0.8 is changed to 0.9 by adding the random value 0.1 to 0.8.

### 1.1.7 A few theoretical results

In this section a few important theoretical results concerning genetic algorithms and optimization algorithms in general are presented.

#### Schema Theorem

One of the most important theoretical results in the area of genetic algorithms is the *Schema Theorem*. The schema theorem applies to genetic algorithms with binary representation. In order to understand the Schema Theorem the concept of *schemas* and some related concepts have to be explained.

A *schema* is very similar to a chromosome. The difference is that the schema may have some “don’t care” symbols usually denoted by * in some positions. An example of a schema could be 001*01*. This schema matches the chromosomes

001*0*01*0*, 001*0*01*1*, 001*1*01*0*, and 001*1*01*1*. Positions not marked by * are called *defining positions*.

The *defining length* of a schema is the distance between the leftmost and the rightmost defining positions. The defining length of the previous example is five. The *order* of a schema is the number of defining positions. The schema in the example has five defining positions and thus an order of five.

What the Schema Theorem says is the following:

**Schema Theorem**
Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm [1].

Or in mathematical terms:

$$\xi(S,t+1) \geq \xi(S,t) \cdot eval(S,t)/(F(t)/pop\_size) \left[1 - p_c \cdot \frac{\delta(S)}{m-1} - o(S) \cdot p_m\right]$$

This probably needs a little explanation:
$\xi(S,t+1)$ is the number of strings (chromosomes) matched by the schema $S$ at time $t+1$. $eval(S,t)$ is the average fitness of the strings in the population matched by the schema $S$ at time $t$. $F(t)$ is the total fitness of the population and thus $F(t)/pop\_size$ is the average fitness of individuals in the population.

$p_c$ is the crossover probability, $\delta(S)$ is the defining length of the schema $S$, $m-1$ is the number of possible crossover sites. Thus the term $1 - p_c \cdot \frac{\delta(S)}{m-1}$ is the probability that the schema will not be destroyed by crossover.

The schema can also be destroyed by mutation. $o(S)$ is the order of the schema $S$ and $p_m$ is the mutation probability. Thus $o(S) \cdot p_m$ is the approximate probability of the schema not being destroyed by mutation.

As can be seen from the equation, schemas with a fitness above average, short defining lengths and low order are increasingly represented in the population.

**Building Block Hypothesis**

The Building Block Hypothesis states the following:

> Genetic algorithms works by combining schemas with low-order building blocks to equally good or better schemas with high-order building blocks.

As implied by the name, the Building Block Hypothesis is just a hypothesis and has not been proved. If the Building Block Hypothesis is true, one should try to

[1]From [Mic95] page 53.

use a representation which makes it possible to have good, short building blocks in the chromosome.

### Deceptive problems

The Building Block Hypothesis states that a genetic algorithm works by combining low-order building blocks to high-order building blocks. A group of problems with the property that the low-order building blocks cannot be combined to high-order building blocks are called *deceptive problems*. The low-order building blocks instead leads the genetic algorithm to a sub-optimal solution and the genetic algorithm is therefore decepted, hence the name deceptive problems.

It has to be noted that almost all deceptive problems encountered in the literature are artificial in the sense that they are constructed to be deceptive. It still remains an open question whether many real-life problems are deceptive or not.

The existence of deceptive problems just emphasizes how important it is to find good building blocks. That is, to use a proper representation when using genetic algorithms. Often it improves performance to tailor the representation and operators like crossover to the problem at hand. The No Free Lunch Theorem presented hereafter suggests that an adaptation of the genetic algorithm to the problem is a good idea.

### The No Free Lunch Theorem

In late 1995 Wolpert and Macready published the article called “No Free Lunch Theorems for Search” [WM95]. In the article it is shown that *all* search algorithms have the *same* performance when averaged over *all* fitness functions. This means that for example genetic algorithms have the same performance as random search when averaged over all fitness functions.

This does not prevent genetic algorithms from being good at finding solutions to certain types of problems. If this is the case, the theorem states that genetic algorithms have correspondingly bad performance on some other types of problems in order to have the same performance as random search on all problems.

The theorem says that a general optimizer cannot have better performance than random search on all problems. This suggests that one should try to specialize the optimizer to a specific class of problems with the tradeoff that there is some other class of problems on which the specialized optimizer will have bad performance.

## 1.2 Constrained optimization problems

This section gives a definition of constrained optimization problems[2] and some examples of constrained optimization problems. Furthermore real-life constrained optimization problems are discussed briefly.

### 1.2.1 Definitions

**Definition 1 (Constrained Optimization Problems)**
Optimize [3]

$$f(\vec{x}), \vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$$

with *bounds*

$$l(i) \leq x_i \leq u(i), 1 \leq i \leq n$$

and *constraints*

$$\begin{array}{lll} g_j(\vec{x}) \leq 0 & \text{for } j = 1, \ldots, q & \text{and} \\ h_j(\vec{x}) = 0 & \text{for } j = q+1, \ldots, m & \end{array}$$

In other words, we have a function $f$ defined in an $n$-dimensional space $\mathbb{R}^n$. The bounds define a $n$-dimensional rectangle in $\mathbb{R}^n$ called the *search space* $\mathcal{S} \in \mathbb{R}^n$. The constraints are divided into inequalities given by $g$ and equalities given by $h$. The constraints define the *feasible* area $\mathcal{F} \in \mathcal{S}$ of the search space. □

**Definition 2 (Active constraints)**
The constraint $g_j$ is said to be *active* in a point $\vec{x}$ in the search space $(S)$, if

$$g_j(\vec{x}) = 0$$

Of course all $h_j$ are also active at all points in the feasible area of the search space. □

**Definition 3 (Feasible and infeasible areas)**
A vector $\vec{x}' \in \mathcal{S}$ is *feasible* if it satisfies the constraints, ie if

$$\begin{array}{lll} g_j(\vec{x}') \leq 0 & \text{for } j = 1, \ldots, q & \text{and} \\ h_j(\vec{x}') = 0 & \text{for } j = q+1, \ldots, m & \end{array}$$

The problem ahead is finding an optimal *and* feasible solution to $f$. Another class of problems are called *constrained satisfaction problems* where the problem is to find a feasible solution.

[2]COP is sometimes used as an abbreviation of for Constrained Optimization Problem in the remainder of this thesis.

[3]With a maximization problem larger function values are considered better and with a minimization problem smaller function values are considered better respectively

### 1.2.2 Example

Minimize

$$G6(\vec{x}) = (x_1 - 10)^3 + (x_2 - 20)^3$$

with bounds

$$13 \leq x_1 \leq 100 \text{ and } 0 \leq x_2 \leq 100$$

and constraints

$$(x_1 - 5)^2 + (x_2 - 5)^2 - 100 \geq 0 \text{ and } -(x_1 - 6)^2 - (x_2 - 5)^2 + 82.81 \geq 0$$

### 1.2.3 Real-life constrained optimization problems

Optimization problems from real-life are very often constrained in one way or another. The scheduling problem is an example of a real-life constrained optimization problem. For example one could consider the time-table of a hospital.

With real life problems it is often useful to distinguish between *hard constraints* and *soft constraints*.

- A hard constraint is a constraint that *must* be satisfied. An example of a hard constraint could be that the doctors only can perform one operation at a time. Another example could be that nurses cannot work for 48 straight hours.
- A soft constraint is a constraint that we would *like* to be satisfied. But the constraint does not *have* to be satisfied and is therefore termed a soft constraints. As an example of a soft constraint one could consider the situation where a doctor would like to have every second weekend off. This constraint is sometimes violated in order to satisfy the hard constraints or some of the soft constraints with higher priority than the soft constraint at hand.

Not all the techniques presented in this thesis are equally suitable for constrained optimization problems with hard and soft constraints. This issue will be discussed in the chapters describing the three techniques.

## 1.3 The test cases

In [MS96] eleven test cases were proposed. Out of the eleven test cases eight were selected as benchmark problems for the techniques presented in this thesis. The eight test cases are the constrained optimization problems with inequalities and there are thus *no* test cases with equalities. All selected test cases are tested during a huge number of runs and these runs require significant computing power and time to analyze the results. Therefore the techniques have not been tested on the four test cases involving equalities but this is an interesting subject for further research.

The test cases have quite different properties when it comes to the number of parameters, the ratio between the feasible part of the search space and the search space, the topology of the search space and so on. The problems will now be presented and some properties of the test cases will be discussed.

### 1.3.1 Test function G1

This function was first presented in [FP90] (test problem 3, pp. 8-9).
Minimize

$$G1(\vec{x}) = 5x_1 + 5x_2 + 5x_3 + 5x_4 - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$$

Constraints

$$2x_1 + 2x_2 + x_{10} + x_{11} \leq 10 \qquad 2x_1 + 2x_3 + x_{10} + x_{12} \leq 10$$
$$2x_2 + 2x_3 + x_{11} + x_{12} \leq 10$$
$$-8x_1 + x_{10} \leq 0 \qquad -8x_2 + x_{11} \leq 0 \qquad -8x_3 + x_{12} \leq 0$$
$$-2x_4 - x_5 + x_{10} \leq 0 \quad -2x_6 - x_7 + x_{11} \leq 0 \quad -2x_8 - x_9 + x_12 \leq 0$$

Bounds
$0 \leq x_i \leq 1$ for $i = 1, \ldots, 9$
$0 \leq x_i \leq 100$ for $i = 10, 11, 12$
$0 \leq x_{13} \leq 1$

Global minimum at

$$\vec{x'} = (1,1,1,1,1,1,1,1,1,3,3,3,1)$$

with $G1(\vec{x'}) = -15$. Six of the nine constraints are active at the global minimum.

### 1.3.2 Test function G2 - Keane's Bump

The test function Keane's Bump is named after Andy Keane.

Maximize

$$G2(\vec{x}) = \frac{|\sum_{i=1}^{n} cos^4(x_i) - 2\prod_{i=1}^{n} cos^2(x_i)|}{\sqrt{\sum_{i=1}^{n} ix_i^2}}$$

Constraints $\prod_{i=1}^{n} x_i \geq 0.75$, $\sum_{i=1}^{n} x_i \leq 7.5n$

Bounds $0 \leq x_i \leq 10$ for $1 \leq i \leq n$

The global maximum is unknown, but values of 0.8036 have been reported.

### 1.3.3 Test function G4

This problem is presented in [Him72] and was originally published in an IBM publication from 1968. Notice that [MS96] has a misprint, where the constant 0.0006262 was erroneously printed as 0.00026. This error was discovered by Michalewicz and Kozieł in experiments with the decoder approach. The decoder approach found feasible solutions better than the global minimum and thus the misprint was discovered!

Minimize

$$G4(\vec{x}) = 5.3578547x_3^2 + 0.8356891x_1x_5 + 37.293239x_1 - 40792.141$$

Constraints
$0 \leq 85.334407 + 0.0056858x_2x_5 + 0.0006262x_1x_4 - 0.0022053x_3x_5 \leq 92$
$90 \leq 80.51249 + 0.0071317x_2x_5 + 0.0029955x_1x_2 + 0.0021813x_3^2 \leq 110$
$20 \leq 9.300961 + 0.0047026x_3x_5 + 0.0012547x_1x_3 + 0.0019085x_3x_4 \leq 25$

Bounds $78 \leq x_1 \leq 102$, $33 \leq x_2 \leq 45$, $27 \leq x_i \leq 45$ for $i = 3,4,5$.

Global minimum at

$$\vec{x'} = (78.0, 33.0, 29.995, 45.0, 36.776)$$

with $G4(\vec{x'}) = -30665.5$. Two out of the six constraints are active at the global minimum.

### 1.3.4 Test function G6

Minimize

$$G6(\vec{x}) = (x_1 - 10)^3 + (x_2 - 20)^3$$

Constraints
$(x_1 - 5)^2 + (x_2 - 5)^2 - 100 \geq 0$
$-(x_1 - 6)^2 - (x_2 - 5)^2 + 82.81 \geq 0$

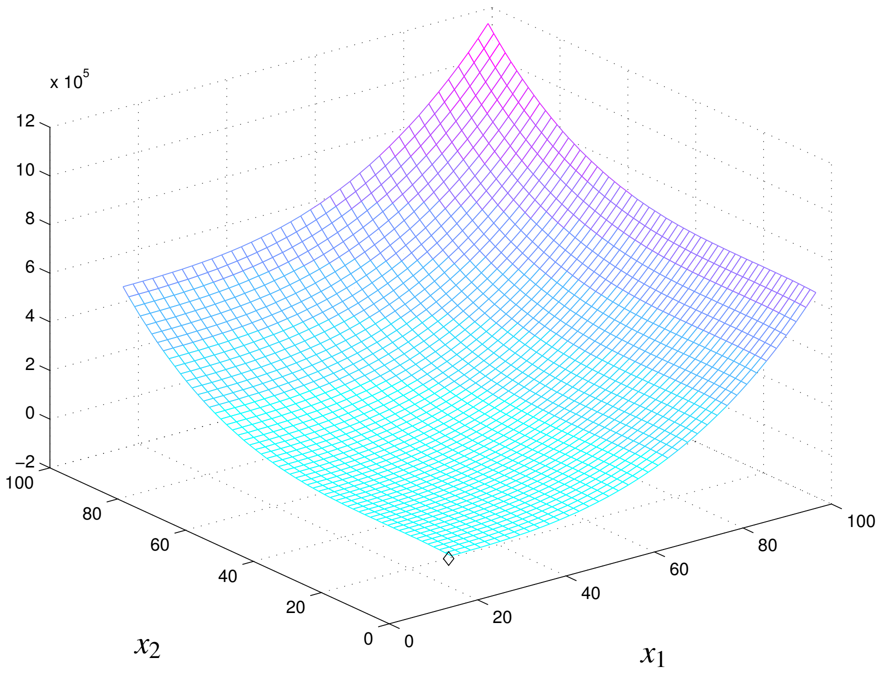

Figure 1.6: Plot of G6

Bounds $13 \leq x_1 \leq 100$ and $0 \leq x_2 \leq 100$

Global minimum

$$\vec{x'} = (14.095, 0.84296)$$

with $G6(\vec{x'}) = -6961.81381$. Both constraints are active at the global minimum. Figure 1.6 shows a plot of G6 with a circle marking the global minimum. The plot was done using 1000 points from the search space and only one point was feasible. Therefore it is not possible to distinguish the feasible area from the infeasible area in the plot.

### 1.3.5 Test function G7

Minimize

$$\begin{aligned} G7(\vec{x}) &= x_1^2 + x_2^2 + x_1x_2 - 14x_1 - 16x_2 + (x_3-10)^2 + 4(x_4-5)^2 + (x_5-3)^2 + \\ &\quad 2(x_6-1)^2 + 5x_7^2 + 7(x_8-11)^2 + 2(x_9-10)^2 + (x_{10}-7)^2 + 45 \end{aligned}$$

Constraints

$$105 - 4x_1 - 5x_2 + 3x_7 - 9x_8 \geq 0$$
$$-3(x_1-2)^2 - 4(x_2-3)^2 - 2x_3^2 + 7x_4 + 120 \geq 0$$
$$-10x_1 + 8x_2 + 17x_7 - 2x_8 \geq 0 \qquad -x_1^2 - 2(x_2-2)^2 + 2x_1x_2 - 14x_5 + 6x_6 \geq 0$$
$$8x_1 - 2x_2 - 5x_9 + 2x_{10} + 12 \geq 0 \qquad -5x_1^2 - 8x_2 - (x_3-6)^2 + 2x_4 + 40 \geq 0$$
$$3x_1 - 6x_2 - 12(x_9-8)^2 + 7x_{10} \geq 0$$
$$-0.5(x_1-8)^2 - 2(x_2-4)^2 - 3x_5^2 + x_6 + 30 \geq 0$$

Bounds $-10.0 \leq x_i \leq 10.0$, $i = 1,\dots,10$

Global minimum

$$\begin{aligned}\vec{x'} &= (2.171996, 2.363683, 8.773926, 5.095984, 0.9906548,\\ &\quad 1.430574, 1.321644, 9.828726, 8.280092, 8.375927)\end{aligned}$$

with $G7(\vec{x'}) = 24.3062091$. Six of the eight constraints are active at the global minimum.

### 1.3.6 Test function G8

Maximize

$$G8(\vec{x}) = \frac{\sin^3(2\pi x_1)\cdot\sin(2\pi x_2)}{x_1^3\cdot(x_1+x_2)}$$

Constraints $x_1^2 - x_2 + 1 \leq 0$ and $1 - x_1 + (x_2-4)^2 \leq 0$

Bounds $0 \leq x_1 \leq 10$ and $0 \leq x_2 \leq 10$

The global maximum is 0.095825 at the point (1.228024, 4.245324). This problem is the easiest problem to solve for all three techniques presented in this thesis. The function G8 is plotted in figure 1.7. The global minimum is marked with a circle.

### 1.3.7 Test function G9

Minimize

$$\begin{aligned}G9(\vec{x}) &= (x_1-10)^2 + 5(x_2-12)^2 + x_3^4 + 3(x_4-11)^2 +\\ &\quad 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7\end{aligned}$$

Constraints

$$127 - 2x_1^2 - 3x_2^4 - x_3 - 4x_4^2 - 5x_5 \geq 0$$
$$282 - 7x_1 - 3x_2 - 10x_3^2 - x_4 + x_5 \geq 0$$
$$196 - 23x_1 - x_2^2 - 6x_6^2 + 8x_7 \geq 0$$
$$-4x_1^2 - x_2^2 + 3x_1x_2 - 2x_3^2 - 5x_6 + 11x_7 \geq 0$$

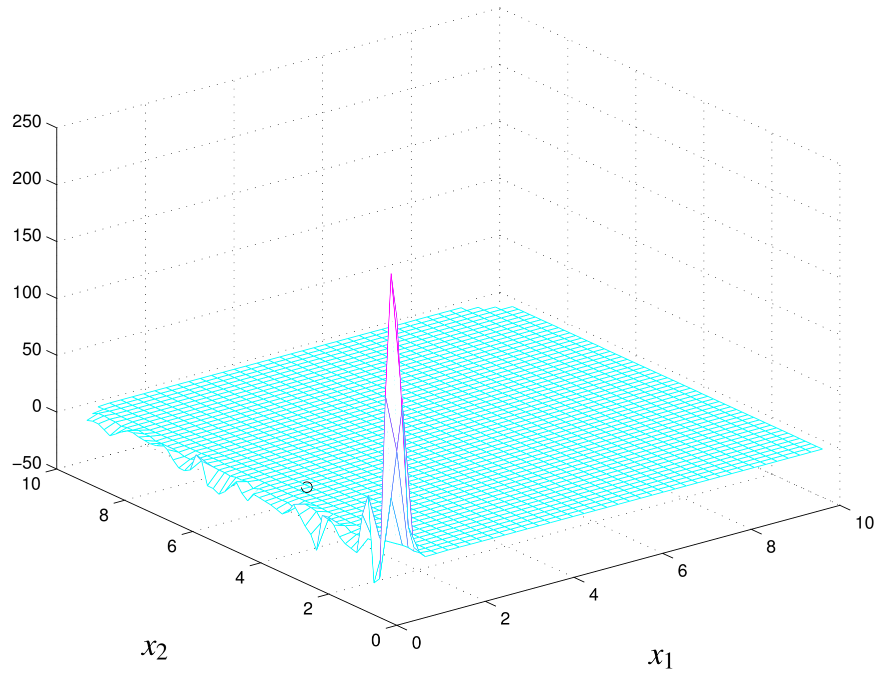

Figure 1.7: Plot of G8

Bounds $-10.0 \leq x_i \leq 10.0$, $i = 1, \ldots, 7$.

Global minimum

$$\begin{aligned}\vec{x'} &= (2.330499, 1.951372, -0.4775414, 4.365726, \\ &\quad -0.6244870, 1.038131, 1.594227)\end{aligned}$$

with $G9(\vec{x'}) = 680.6300573$. Two out of the four constraints are active at the global minimum.

### 1.3.8 Test function G10

Minimize

$$G10(\vec{x}) = x_1 + x_2 + x_3$$

Constraints
$1 - 0.0025(x_4 + x_6) \geq 0$, $1 - 0.0025(x_5 + x_7 - x_4) \geq 0$
$1 - 0.01(x_8 - x_5) \geq 0$, $x_1 x_6 - 833.33252 x_4 - 100 x_1 + 83333.333 \geq 0$
$x_2 x_7 - 1250 x_5 - x_2 x_4 + 1250 x_4 \geq 0$, $x_3 x_8 - 1250000 - x_3 x_5 + 2500 x_5 \geq 0$

| Function | Ratio |
|---|---|
| G1 | 0.000180 |
| G2 | 99.996964 |
| G4 | 26.950348 |
| G6 | 0.006760 |
| G7 | 0.000110 |
| G8 | 0.864150 |
| G9 | 0.523310 |
| G10 | 0.000560 |

Table 1.1: The ratio of feasible points in %

Bounds
$100 \leq x_1 \leq 10000$, $1000 \leq x_i \leq 10000$, $i = 2,3$, $10 \leq x_i \leq 1000$, $i = 4,\ldots,8$

Global minimum

$$\begin{aligned}\vec{x'} &= (579.3167, 1359.943, 5110.071, 182.0174,\\ &\quad 295.5985, 217.9799, 286.4162, 395.5979)\end{aligned}$$

with $G10(\vec{x'}) = 7049.330923$. All constraints are active at the global minimum. This problem is the hardest problem to solve for all three techniques. In the next section the penalty approach is presented and this approach did in fact very seldom find a feasible solution.

## 1.3.9 Description of the problems

In order to give an impression of the problems, 10 million random points where dispersed in the domain of each problem. This was used to calculate the ratio of feasible points in the search space. The ratios are displayed in table 1.1.

To give an idea of the hardness of the test problems, the number of dimensions, the number of linear inequalities in the constraints and the number of non-linear constraints in the constraints are listed in table 1.2. The table also contains information about the number of active constraints at the optimum, listed in the last column.

| Function | Dimensions | Linear inequalities | Nonlinear inequalities | AC |
|---|---|---|---|---|
| G1 | 13 | 9 | 0 | 6 |
| G2 | 20 | 0 | 2 | 1 |
| G4 | 5 | 0 | 6 | 1 |
| G6 | 2 | 0 | 2 | 2 |
| G7 | 10 | 3 | 5 | 6 |
| G8 | 2 | 0 | 2 | 0 |
| G9 | 7 | 0 | 4 | 2 |
| G10 | 8 | 3 | 3 | 6 |

Table 1.2: Properties of the test functions.

# Chapter 2

# Penalized Fitness Functions for COPs

Penalty
— *the suffering or the sum to be forfeited to which a person agrees to be subjected in case of nonfulfillment of stipulations*
Merriam-Webster's Collegiate Dictionary

## 2.1 Introduction

The search space $\mathcal{S}$ for a constrained optimization problem consists of two parts: the feasible region(s) denoted by $\mathcal{F}$ and the infeasible region(s) denoted by $I$.

A genetic algorithm will end up with individuals representing points in the feasible regions and the infeasible regions. The question is: How do we evaluate these individuals? Is an individual which violates some constraints but has the better function value of 1000 fitter than an individual which does not violate any constraints but has a function value of only 900?

Several methods which handle this problem has been proposed. One of the simplest methods is to *penalize* infeasible individuals. In other words, an infeasible individual will receive some kind of penalty for not satisfying the constraints:

$$f_{pen}(\vec{x}) = f_{obj}(\vec{x}) \pm P(\vec{x})$$

The fitness of each individual is the combined values of the objective function and the penalty function. The value of the penalty function will be added if the constrained optimization problem is a minimization problem and subtracted otherwise. In the remainder of the chapter it will be assumed that the problems are minimization problems and only fitness functions of the former type are described.

What should the penalty function be like? One possibility is to make the penalty function depend on how hard it is to repair an infeasible individual. An infeasible individual is repaired by changing it in some way to obtain a feasible individual. Another possibility is to let the penalty function depend on the size of the constraint violation. If for example one of the constraints is $x_1 \cdot x_2 \leq 0$ and the individuals $i_1 = (1,2)$ and $i_2 = (10,5)$ are both infeasible, it does not seem unfair to give a smaller penalty to $i_1$ than to $i_2$.

The second method is the one investigated in this chapter. In more general terms the fitness function is:

$$f_{pen}(\vec{x}) = f_{obj}(\vec{x}) + W \cdot P'(\vec{x})$$

where $P'$ is the sum of constraint violations and $W$ is a weight scalar called the *penalty weight* in the remainder of this chapter.

It is interesting to see how easily the penalty approach can be extended to handle constrained optimization problems with hard and soft constraints.

## 2.2 The penalty weight

The penalty weight is one of many parameters in the genetic algorithm. In [EHM98] methods which use adaptation of parameters are grouped into different kinds of categories according to the type of adaptation which is taking place.

- Constant parameters
- Dynamic parameters
  - Deterministic changing parameters
  - Adaptive parameters
  - Self-adaptive parameters

With respect to the penalty weight this gives several possibilities. $W$ may be constant during evolution (constant penalty weights) or dynamic, ie changing during evolution. The dynamic weight may be achieved by altering the weight by some deterministic rule (deterministic changing penalty weights), by using feedback from the search to alter the weight (adaptive penalty weights), or by incorporating the weight into each individual and thereby making it subject to mutation and crossover (self-adaptive penalty weights).

Whereas there are several examples of constant, deterministic changing, and adaptive penalty weights in literature, there are no examples of self-adaptation of the penalty weights.

The purpose of this chapter is to investigate the new approach of self-adapting the penalty weights, and to compare it to the approaches of constant penalty weights, deterministic changing penalty weights, and adaptive penalty weights.

## 2.3 The parameter schemes

The four different parameter schemes will now be described in the way they have been used in the system built for testing.

### 2.3.1 Constant penalty weights

Each individual is evaluated using

$$f_{pen}(\vec{x}) = f_{obj}(\vec{x}) + W \cdot P'(\vec{x})$$

where $W$ is constant during the run and the same for all individuals. This is probably the most commonly used way to handle constraints when using genetic algorithms. The idea is very simple and the performance on most problems is adequate.

### 2.3.2 Deterministic changing penalty weights

As with constant penalty weight, each individual is evaluated using

$$f_{pen}(\vec{x}_i) = f_{obj}(\vec{x}) + W(t) \cdot P'(\vec{x})$$

and W(t) is the same for all individuals. But W(t) may now change over time. One example could be that the weight increases linearly over time, e.g. $W(t) = t$. Another example could be that the weight decreases linearly over time, e.g. $W(t) = -t + 10000$. A third possibility could be that the weight increases exponentially over time e.g. $W(t) = e^{0.0001 \cdot ln(8000) \cdot t} + \frac{1}{5}t$. The plots of the functions in the three examples can be seen in figure 2.1.

Examples of the use of the deterministic changing penalty weights can be found in [JH94] and [KP98]. In [JH94] the functions G6, G9, G10 and a fourth function are used as test cases and good results are reported especially in the case of G10.

### 2.3.3 Adaptive penalty weights

The idea is to use feedback from the search to adapt the penalty weight. All individuals are evaluated using the same weight. This could be achieved by using the

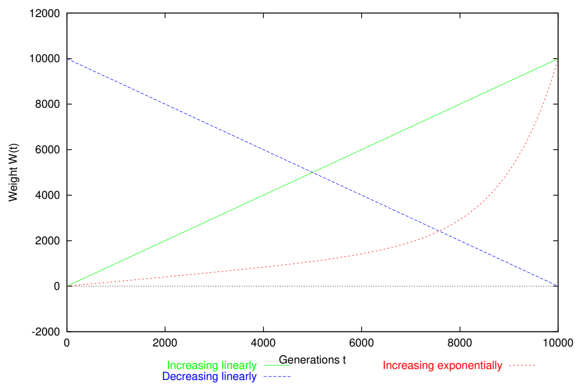

Figure 2.1: Deterministic changing penalty weights

ratio of feasible individuals in the population. If the ratio of feasible individuals in the population is larger than for example 15%, we would increase the penalty weight, or decrease it, if the ratio is less than 15%. In more general terms:

$$W = \begin{cases} W + \delta & \text{if } r > r_0 \\ W - \delta & \text{otherwise} \end{cases}$$

where $r = \frac{I}{F+I}$ is the current ratio feasible individuals in the population and $r_0$ the threshold value for the ratio.

In the current system the penalty weight is updated after each generation but it is of course possible to update the penalty weight after every $n$th generation where $n$ is an arbitrary number.

Another way of adapting the penalty weight can be found in [EvdH97]. In this article a penalty weight *vector*[1] is adapted. The penalty weight vector contains one entry for each constraint and thereby gives the possibility to have different penalty weights for different constraints. The weights in the penalty weight vector are adapted by looking at the fittest individual and increasing the values of the penalty weights for the constraints that the fittest individual violates. Using this approach the new method outperforms the best heuristic technique on the 3-SAT problem.

## 2.3.4 Self-adaptive penalty weights

The penalty weight is a part of each individual and is therefore subject to mutation and crossover. If we have two individuals $x_1$ and $x_2$ with the penalty weights $W_1$

[1]As opposed to penalty weight *scalar*.

and $W_2$ respectively, and $W_1 > W_2$ the individuals are evaluated using

$$f_{pen}(\vec{x}_i) = f_{obj}(\vec{x}) + \max(W_1, W_2) \cdot P'(\vec{x})$$

If the individuals are assigned a fitness using the weight in the weight genes and two individuals with the same solution but different weight-genes are compared, the one with the lower penalty weight will have the best fitness.

Consider the following example where the individuals represent solutions to the G6 test function. The genes of the first chromosome have the values 14, 2, and 10 for $x_1$, $x_2$ and the penalty weight respectively. The genes of the second chromosome have the values 14, 2, and 5. Notice that the value of the weight gene is the only difference between the two chromosomes. The value of the two constraints $c_1$ and $c_2$ in the point (14, 2) is $c_1 : (14-5)^2 - (2-5)^2 - 100 = -28$ and $c_2 : -(14-6)^2 - (2-5)^2 + 82.81 = 137.81$. As $c_1 \geq 0$ and $c_2 \geq 0$, the first constraint is violated and the second is not. The value of G6 in the point (14, 2) is $(14-10)^3 + (2-20)^3 = -5768$. Using the naive approach the fitness of the first chromosome will be $-5768 + 10 \cdot (|-28| + 0) = -5488$ and the fitness of the second chromosome $-5768 + 5 \cdot (|-28| + 0) = -5628$. The second chromosome has the best fitness and would be selected for reproduction if the two individuals were to compete against each other in a binary tournament.

Thereby, a pressure towards lower penalty weight values will be created and this is highly undesirable. The problem is that the weight directly influences the fitness of the individual and there will thus be a very strong selection pressure for low penalty weights. The low penalty weights will probably not be of great help in finding feasible solutions, as infeasible solutions will receive very small penalties for violating the constraints.

How do we solve this problem? One solution could be to use the largest of the two penalty weights when comparing two individuals. Then there no longer will be a pressure towards lower penalty weights.

Although there are sound arguments for using $\max(W_1, W_2)$, one could use $\min(W_1, W_2)$, $\text{avg}(W_1, W_2)$, or even a completely different function. The results of experiments with different functions will be presented later in this chapter.

## 2.4 The system

The genetic algorithm used in the system has real-valued representation. Each gene in the chromosome represents one variable in the test problem. The test problem G6 is

$$G6(\vec{x}) = (x_1 - 10)^3 + (x_2 - 20)^3$$

and is a 2-dimensional problem. Each chromosome therefore has two genes representing $x_1$ and $x_2$ respectively. The mapping between the genotype and the phenotype is the simplest possible and in fact there is no need to distinguish between the genotype and the phenotype.

Individuals are selected for reproduction using binary tournament selection. New individuals are created from the selected individuals using uniform crossover where each gene is taken from either parent with a probability of 0.5.

Each gene in the newly created individuals is subject to mutation with a probability of $p_m$. If the gene is selected for mutation, a random value is added to the value of the gene. The random value is generated by a random number generator which returns numbers that are normal distributed with a mean of 0.0 and standard deviation 0.5. After addition of the random value[2], the genes are adjusted so that the value of the gene does not exceed the domain of the variable the gene represents.

The genetic algorithms is generational like most other genetic algorithms. With an elite size of one the best individual is copied from the old generation to the new generation. The fittest feasible individual is considered the best individual when it comes to the elite. If there are no feasible individuals in the population, a random individual is copied to the population of the next generation.

The population size was fixed at a value of 30 in all experiments. The value of 30 was chosen for no particular reason other than it did not seem too high or too small.

With the method of self-adapting the penalty weight, each individual in the population has an extra gene containing the value of the penalty weight. The system gives two possible ways of initializing the penalty weight genes. The first possibility is to initialize the penalty weight genes using random numbers from a Gaussian distribution with mean[3] $W$ and standard deviation $\frac{1}{10} \cdot W$. Another possibility is to initialize the penalty weights with random values where the random values are distributed uniformly in the interval from 0 to $2W$. This is done in order to have a mean weight of $W$.

## 2.5 Experiments and results

The four approaches of

1. constant penalty weights - CON
2. deterministic changing penalty weights - DET
3. adaptive penalty weights - ADA

[2]Note that the random value of course can be negative.

[3]Specified as a command line parameter to the program.

4. self-adaptive weights - SAD

will be compared.

All experiments were performed ten times and the average values of these runs were used to compensate for the stochastic nature of the experiments.

As a measure of performance, we will look at the fitness of the fittest feasible individual after 5000 generations.

## 2.5.1 Tuning the parameters

Parameters like the crossover rate and mutation rate have a great influence on the performance of the genetic algorithm. A number of experiments were conducted to tune the crossover rate and the mutation rate. The crossover rate had values of 0.00, 0.01, 0.10, 0.50, and 1.00. The mutation rate also had values of 0.00, 0.01, 0.10, 0.50, and 1.00. This amounts to 25 different settings of the mutation rate and the crossover rate. For each of the four penalty weight schemes, a number of experiments were conducted:

CON The 25 experiments were repeated with a weight of 1, 10 and 100.

DET The 25 experiments were repeated with various different start and stop values and with different ways of changing the weight. The different values can be found in table 2.1.

| Weight change | Start | Stop |
|---|---|---|
| Linear | 0 | 100 |
| Linear | 0 | 1000 |
| Linear | 100 | 0 |
| Linear | 1000 | 0 |
| Exponential | 0 | 100 |
| Exponential | 0 | 1000 |

Table 2.1: Parameter setting for the deterministic changing penalty weight

ADA The adaptive weight scheme was tuned using the following parameters: The threshold ratio of feasible individuals could have values of 0.20, 0.50 and 0.80. The weight could be changed in an absolute way with a change of weight of 1 or 10. Alternatively the weight could change relatively to its current value by 1% or 10%.

SAD The self-adaptive penalty weight was tuned using three different value for the weight (1, 10, and 100) and with uniform and normal distribution of the

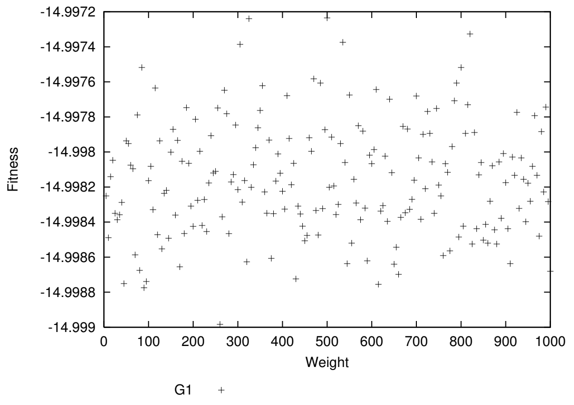

Figure 2.2: Weight used in the CON approach vs. fitness - G1

initial weights. This gives a total of 6 experiments repeated 25 times for each of the crossover and mutation rates.

With these experiments it was possible to find some good settings for the mutation rate and the crossover for each of the four penalty weight schemes. All the following experiments were conducted with these values.

Of the 25 possible combinations of mutation rates and crossover rates, the one with a mutation rate of 0.1 and a crossover rate of 1.0 was by far the most successful. A mutation rate of 0.1 seems to be a good compromise between the probably too low value of 0.01 and the too high value 0.5. What the crossover rate concerns, the value of 0.5 was probably too low and therefore the value of 1.0 was a clear winner.

### 2.5.2 Fine tuning the parameters

In order to achieve good performance, some additional fine tuning was done for each of the four ways of changing the penalty weight.

For all eight benchmark problems the constant penalty weight was fine tuned using the weights $0, 5, 10, ..., 990, 995, 1000$. For the functions G1, G2, G7, and G9 the value of the penalty weight did not seem to make a difference. For the function G4 the weight had to be larger than 820, for G6 larger than 300 and for G8 larger than 50. Figures 2.2 and 2.3 show plots of G1 and G6 with the weight along the $x$-axis and the fitness of the fittest feasible individual up the $y$-axis.

DET was fine tuned using all combinations of 0.1, 0.5, 1, 5, 10, 50, 100, 500, 1000, and 5000 as the start and stop weights respectively. This amounts to 100 experiments for each of the two ways of changing the weight (linearly and exponentially). The experiments show that there does not seem to be any general

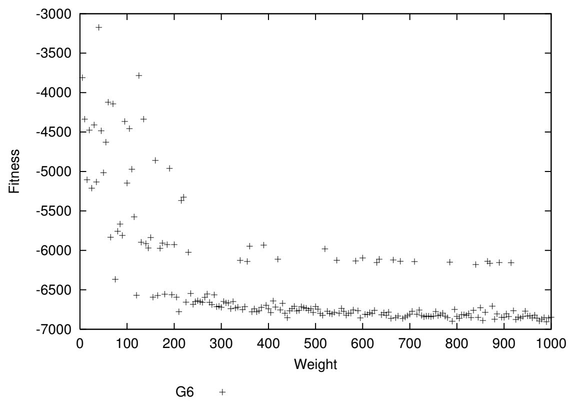

Figure 2.3: Weight used in the CON approach vs. fitness - G6

rules regarding the start and stop values. For example, the best value for G2 was obtained using 100 as the start weight and 1000 as the value of the weight at the end of the run. For G4 the best performance was achieved with a start weight of 1000 and an end value of 0.1. The start and stop values therefore have to be tuned to the problem at hand. With respect to the two ways of changing the weight, linearly and exponentially, the linear approach had the best performance for all test functions. The exponentially changing weight seems to be changing too fast for the genetic algorithm to accommodate.

The ADA was fine tuned using the ratios of 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. $\delta$ (the weight change value) had values of 0, 20, 40,...,, 960, 980, and 1000. Finally, the weight could change absolutely or relatively. For G1, G2 and G7 the best value of $\delta$ was 0. With this value the weight does not change at all and it is the same as the constant penalty weight. For the rest of the functions the value of $\delta$ did not seem to have any impact on the performance. For G1, G2, G7, and G9 the value of the ratio did not seem to be important. For G4 and G6 a ratio of 0.0 gave the best results and nothing useful can be said about the value of $\delta$. A 3-dimensional plot of the ratio and the value of $\delta$ versus the fitness is depicted in figure 2.4.

With a ratio value of 0.0, the weight is increased if there are one or more infeasible individuals in the population. The value of 0.0 allows the weight to increase to large values very fast and this is interesting combined with the fact that G4 and G6 gave the best performance with CON when the penalty weight was not too small. For G8 all but large values of the ratio gave good performance. It is quite clear from the plot of G4 that nothing can be said about the value of $\delta$ and that the performance increases with the ratio going towards 0.

To tune the SAD approach, all combinations of initial penalty weights from 0

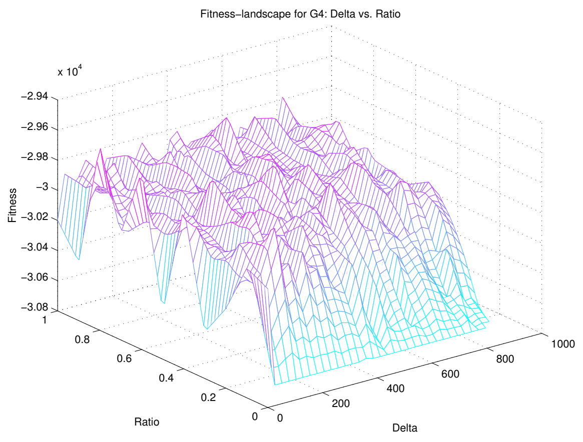

Figure 2.4: Ratio, $\delta$ and fitness for G4

to 1000 in steps of 5 and the two initialization methods, uniform and normal, were tried. Comments on the results are in section 2.5.3.

A lot of experiments have been conducted and table 2.2 shows the performance of each approach. The results were obtained in the following way:

The results from the fine tuning experiments were processed statisticly. This made it possible to rank the parameter settings for all approaches based on the average performance. Let us look at the SAD approach for an example. The fittest feasible individual after 5000 generations had an average fitness of -14.9986 for the test function G1 with a mutation rate of 0.1, a crossover rate of 1.0, and an initial weight of 1 which was normal distributed. For all other parameters tried, there were none which gave better results on G1 when looking a the average performance of the ten runs. In a similar way the best parameters for each approach and each test function was found. With these newly found parameters ten *new* runs were conducted for each approach and for each test function. The results of the runs can be found in table 2.2.

From the ten new runs the best and the worst values are displayed to give an impression of the variance of the results. Furthermore, the average performance in the ten runs is displayed. The best results among the four approaches are marked in **bold** and the worst are marked in *italics*.

It is clear from the table that G8 proved no challenge to any of the penalty weight schemes. Furthermore, it is also quite clear that G10 was too big a challenge for all schemes, as none of them were able to find a number of feasible points high enough to justify a comparison of the schemes.

What else can be concluded from the table? Based on the number of times that each scheme had the best and worst performance respectively, the best performer

| | | CON | DET | ADA | SAD | Optimum |
|---|---|---|---|---|---|---|
| G1 | Best | **-14.9997** | **-14.9997** | *-14.9993* | *-14.9993* | 15.0000 |
| | Average | *-14.9985* | **-14.9990** | -14.9986 | *-14.9985* | |
| | Worst | -14.9971 | **-14.9982** | -14.9979 | *-14.9967* | |
| G2 | Best | *0.747460* | 0.784245 | **0.788575** | 0.782672 | 0.8036 |
| | Average | *0.685595* | 0.732685 | 0.737603 | **0.743063** | |
| | Worst | *0.560413* | **0.670220** | 0.661285 | 0.666392 | |
| G4 | Best | **-30665.5** | -30665.3 | -30664.7 | *-30649.8* | -30665.5 |
| | Average | **-30665.5** | -30664.5 | -30662.9 | *-30584.5* | |
| | Worst | **-30665.4** | -30661.1 | -30659.8 | *-30520.3* | |
| G6 | Best | **-6952.6** | -6922.8 | *-6915.5* | -6948.6 | -6961.8 |
| | Average | -6838.7 | -6893.4 | *-6790.0* | **-6921.0** | |
| | Worst | -6704.0 | -6865.0 | *-5954.8* | **-6891.4** | |
| G7 | Best | *24.988* | 24.880 | 24.713 | **24.597** | -24.306 |
| | Average | *26.786* | **25.755** | 26.299 | 26.099 | |
| | Worst | 29.138 | **27.213** | 28.709 | *30.244* | |
| G8 | Best | 0.095825 | 0.095825 | 0.095825 | 0.095825 | 0.095825 |
| | Average | 0.095825 | 0.095825 | 0.095825 | 0.095825 | |
| | Worst | 0.095825 | 0.095825 | 0.095825 | 0.095825 | |
| G9 | Best | *680.75* | **680.72** | **680.72** | **680.72** | 680.63 |
| | Average | *681.26* | **680.90** | 680.94 | 681.12 | |
| | Worst | *683.24* | 681.45 | **681.18** | 681.63 | |
| G10 | Best | - | - | - | - | 7049.33 |
| | Average | - | - | - | - | |
| | Worst | - | - | - | - | |
| Best | | 5 | 8 | 3 | 5 | |
| Worst | | 9 | 0 | 4 | 6 | |

Table 2.2: Results of the fine tuning runs.

is the approach of deterministic changing penalty weights. The DET approach had the best results in eight out of eighteen possible and *no* worst results. This makes the DET approach a very solid performer.

When it comes to a looser, the title goes to the approach of adapting the penalty weight based on the ratio of feasible individuals in the population. The problem probably is, that the ratio of feasible individuals maybe not always is a good indication of when and by how much to change the penalty weight. Furthermore, it has to be stressed that the performance of the ADA approach is not bad at all. But when compared to the other approaches, they are a little bit better on this particular suite of benchmark problems.

The performance of the methods of constant penalty weights and self-adaptive penalty weight was approximately the same on the benchmark problems. The SAD approach does not have very good performance on the G4 test problem where the CON approach is superior. When the attention is turned to the G2 test function, the picture is reversed. When the results of all test functions for the SAD approach and the CON approach are compared to each other, the SAD approach is slightly better than the CON approach. The performance of the SAD approach is in fact surprisingly good.

### 2.5.3 Further experiments with SAD

#### The initial weights

Some experiments where performed which considered SAD's sensitivity to the initial value of the weight gene.

With the settings found in the tuning experiments, the SAD approach was tried on all test cases with initial weights from 0 to 1000 in steps of 5. Furthermore, two ways of initializing the weight were tried. One way was to initialize the gene with a Gaussian distribution with mean $W$ and standard deviation $\frac{1}{10}W$, giving a mean of $W$ for the initial weights of all individuals. The other way was to initialize the weight genes to values distributed evenly in the interval from 0 to $2W$ again giving a mean of $W$.

The results show that the best performance was obtained when the weights were initialized using the Gaussian distribution.

Furthermore, in most cases (G1, G2, G7, and G9) the performance was not affected by the initial size of the weight. In the rest of the cases (G4, G6, and G8) the weights had to be larger than a certain threshold to obtain optimal performance. Figure 2.5 show plots of functions G1 and G4 with the fitness of the fittest feasible individual along the $y$-axis and the initial weight along the $x$-axis.

It is interesting to note that, for example for G4 the initial weight in SAD had to be larger than 250 and in CON larger than 820 to achieve good performance. SAD seems to be less sensitive to the value of the weight and the weight in fact seems to get adapted.

#### Changing the mutation rate

Another parameter that might affect the performance is the mutation rate. Several experiments were conducted where the mutation rate of the weight gene was changed (the mutation rate of the rest of the genes was 0.1 in all experiments). As with the initial weight experiments, most cases (G1, G6, G7, G8, and G9)

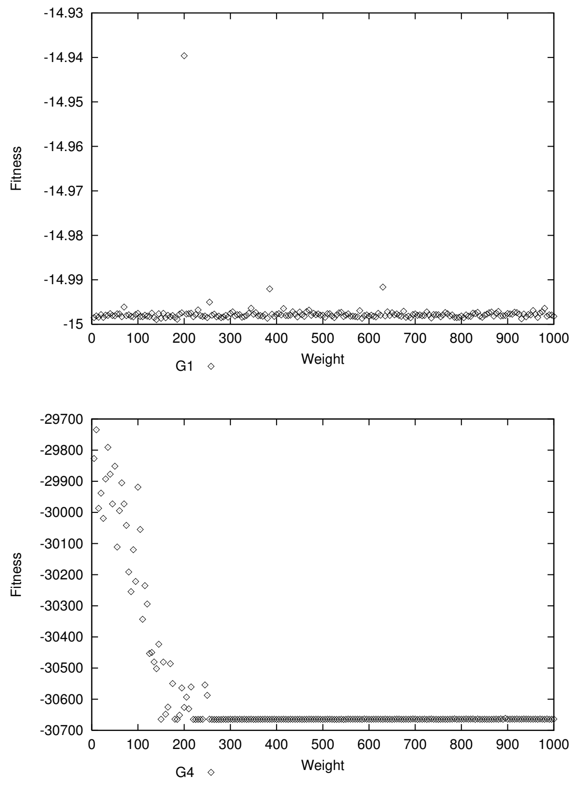

Figure 2.5: Initial weight vs. fitness

where unaffected by changing the parameter. In the case of G2 the performance decreased with increased mutation rate. With G4 the picture was reversed with increased performance with increased mutation rate. Plots of fitness vs. mutation rate for functions G1 and G4 can be seen in figure 2.6.

### Comparing individuals - other approaches

As mentioned in section 2.3.4, the individuals are evaluated using the fitness function:

$$f_{pen}(\vec{x}_i) = f_{obj}(\vec{x}) + \max(W_1, W_2) \cdot P'(\vec{x})$$

But instead of using $\max(W_1, W_2)$, $\min(W_1, W_2)$ or $\text{avg}(W_1, W_2)$ could be used. Some experiments with the two latter options have been performed and the results are reported in this section.

From the tuning experiments some good values for the mutation rate and the crossover rate were found for each function. The fine tuning experiments were

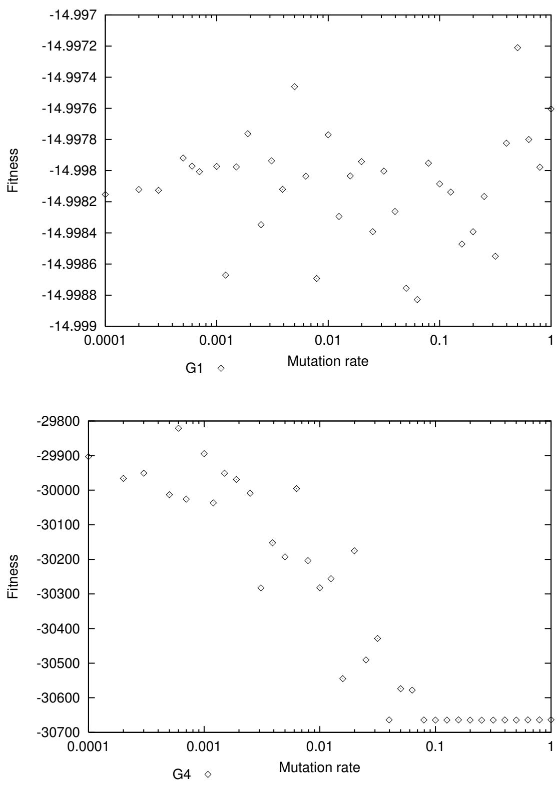

Figure 2.6: Mutation rate vs. fitness

repeated using min and avg in the fitness functions to tune the values of the initial weights and to choose the best way of initializing the weights. The average performance was used in determining the best performing parameters. Using the best performing parameters, ten new runs were performed.

The results are displayed in table 2.3. As in table 2.2, the table contains the best and the worst values from the ten runs. Furthermore, the average performance in the ten runs is displayed. The best results among the three ways of evaluating individuals are marked in **bold** and the worst are marked in *italics*.

It is quite clear from the table, that the fitness function using max gave the best results with the best performance in eight out of nine cases. This is what was expected. Both min and avg had inferior performance with avg having the worst performance on the three selected test functions.

| | | Max | Avg | Min | Optimum |
|---|---|---|---|---|---|
| G1 | Best | -14.9993 | *-14.9983* | **-14.9999** | 15.0000 |
| | Average | **-14.9985** | *-14.9856* | -14.9880 | |
| | Worst | **-14.9967** | *-14.9418* | -14.9524 | |
| G4 | Best | **-30649.8** | -30642.9 | *-30608.1* | 30665.5 |
| | Average | **-30584.5** | -30565.6 | *-30554.3* | |
| | Worst | **-30520.3** | -30513.9 | *-30489.4* | |
| G9 | Best | **680.72** | *680.85* | *680.85* | 680.63 |
| | Average | **681.12** | *681.55* | 681.47 | |
| | Worst | **681.63** | *683.95* | 682.84 | |
| Best | | 8 | 0 | 1 | |
| Worst | | 0 | 6 | 4 | |

Table 2.3: Results of experiments with other SAD methods.

**The development of the penalty weight over time**

Another interesting thing to look at, is how the weights evolve over time. Figure 2.7 shows average values of the weight gene over the course of a run for the function G1. For the functions G1, G4, G6, G7, and G9 the picture is the same, the weight value gets larger over time. For the functions G2 and G8 the average weight value gets neither larger nor smaller.

## 2.6 Conclusions

The idea of self-adaptation is indeed very beautiful. In the experiments conducted, the self-adaptive approach showed quite robust performance, but could not outperform the other approaches when carefully fine tuned. The self-adaptive approach might come in handy in experiments where there is no or little time for fine tuning, or experiments where one wants to achieve good performance but not necessarily the best performance.

## 2.7 Future work

### 2.7.1 Penalty weight vector

Several interesting topics remain to be investigated. First of all, it would be interesting to see what happens if a weight *vector* instead of a weight *scalar* is adapted. Will it improve performance because of the possibility of adapting the weight for

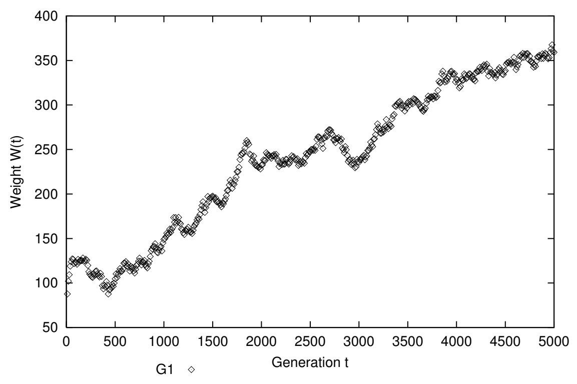

Figure 2.7: Weight over time

each constraint or will the added complexity of adapting a vector degrade the performance?

### 2.7.2 Self-adaptation

Another interesting aspect is to investigate how the self-adaptation of the penalty weight will interact with self-adaptation of other parameters such as the mutation rate and the crossover rate.

### 2.7.3 Real-life problems

As mentioned in the introduction, real-life problems often requires the algorithm to handle hard and soft constraints. The question is how to handle hard and soft constraints using the penalty approach. One possible solution is simply to use different penalty weights for the hard and soft constraints. When the hard constraints are violated the penalty should be so high that the algorithm will produce a solution which does not violate any of the hard constraints. The violation of the soft constraints in the final solution should be minimal but does not have to be zero.

Another topic somewhat linked to that of the hard and soft constraints, is that among the soft constraints one may give different priorities to the different constraints. For example one could imagine that it is more important for a nurse to have the first two weeks in July off, than having every second weekend off all year. The first constraint should therefore be given a higher priority than the second. Again this can be quite easily achieved using the penalty approach. Each constraint should have a corresponding penalty weight and the size of the penalty weight should reflect the priority of the constraint. This is easily done for the constant penalty weight approach. For the self-adaptive penalty weight approach the same might be achieved by initializing the weight genes with values reflecting the priority of each constraint. But from there on it is not quite clear how the penalty weights will change and this too could be an interesting topic for future research.

To summarize the penalty approach is quite suitable when it comes to the solution of real-life problems.

# Chapter 3

# Ant Colony Systems for COPs

*Ants certainly communicate information to each other,*
*and several unite for the same work, or for games of play.*
Charles Darwin, The Descent of Man

## 3.1 Ant colonies in nature

In nature ants spend a lot of time foraging for food. Biologists have made experiments with ants and have discovered that ants have the ability to find the shortest route from the nest to the food source. But how do they do it — do the ants possess some kind of intelligence? The answer is that the ability to find the shortest route is an *emergent behavior* of the ants. When an ant walks around, it leaves a trail of pheromone. Pheromone is a chemical, that ants can smell. The ants choose which way to go depending on the amount of pheromone in each direction, and will choose the direction with the strongest pheromone smell. This very simple behavior allows ants to collectively find the shortest route from the nest to the food source.

To see how this can happen, let us take a look at an example. The example is depicted in figure 3.1 and table 3.1, and explained in the following.

At time $t = 0$, 40 ants start from the nest. At $t = 1$ they reach A, where they have to decide to go left (AB1) or right (AB2). As there is no pheromone, they decide randomly and therefore 20 choose to take AB1 and 20 take AB2. At $t = 5$ the 20 ants which took AB1 arrive at B where they head for the food. At $t = 7$ they are at B again and so are the 20 ants which took AB2. The food-carrying ants now have to choose whether to take AB1 or AB2 based on the amount of pheromone. As both AB1 and AB2 have been traveled by the same number of ants, the amount of pheromone on each path is the same and the ants choose randomly with 10 ants taking AB1 and AB2 respectively.

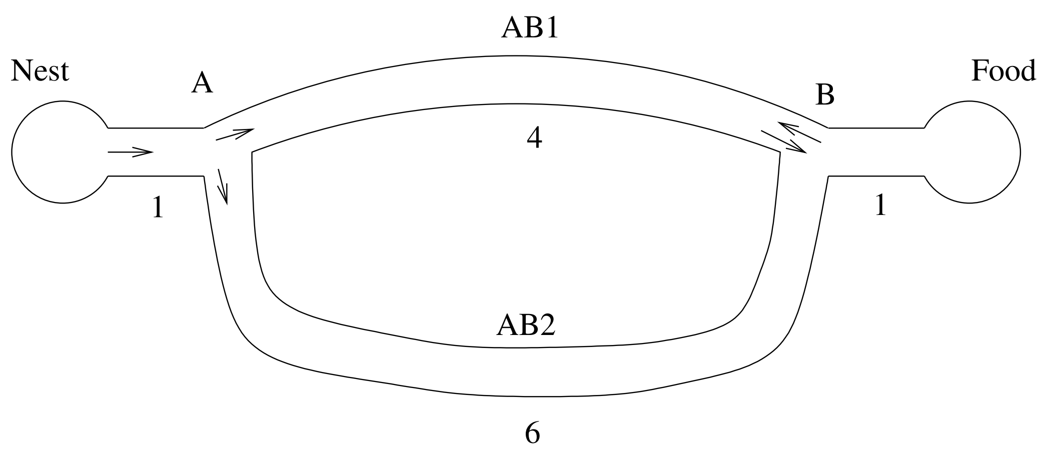

Figure 3.1: Simplified example of ants foraging for food

| t | A | B | AB1 | AB2 | Food | Nest |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 40 |
| 1 | 40 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 20 | 20 | 0 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 5 | 0 | 20 | 0 | 20 | 0 | 0 |
| 6 | 0 | 0 | 0 | 20 | 20 | 0 |
| 7 | 0 | 40 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 10 | 10 | 20 | 0 |
| 9 | 0 | 20 | 10 | 10 | 0 | 0 |
| 10 | 0 | 0 | 20 | 20 | 0 | 0 |
| 11 | 10 | 0 | 10 | 20 | 0 | 0 |
| 12 | 0 | 0 | 10 | 20 | 0 | 10 |
| 13 | 10 | 0 | 10 | 20 | 0 | 0 |
| 14 | 0 | 0 | 16 | 24 | 0 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 3.1: Table showing the number of ants at each location during time

At $t = 13$ the 10 ants which took AB1 has arrived at the nest, dropped their food and are now at A. At this point 15 ants have dropped pheromone on AB1 and 10 on AB2. Therefore 6 of the 10 ants will choose AB1 and 4 will choose AB2, thus making AB1 the more popular (and the shorter) route. It is not hard to see that this pattern will repeat itself during time and *reinforce* the ants to prefer the shortest

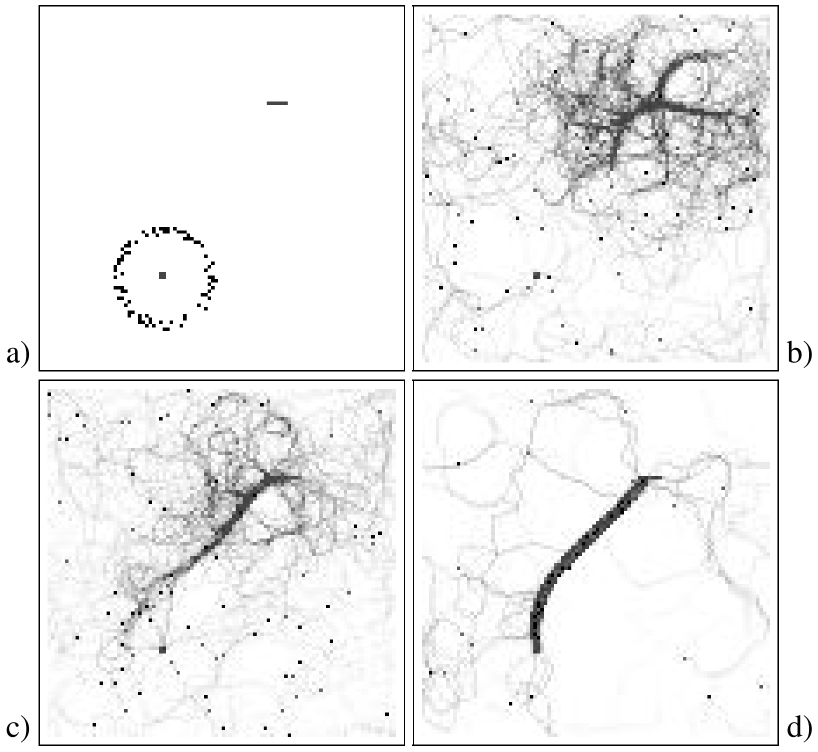

Figure 3.2: Output from SimulAnt.

route AB1.

The system explained in a simplified version above, has been implemented by Mark Wodrich [Wod96] in a system called *SimulAnt*. Figure 3.2 shows output produced by the system.

In a) all ants are gathered at the nest and head out in random directions. In b) there is a plot of some ants which have found food, but do not know where to go since the other pheromone depositing ants essentially move randomly and therefore they too move randomly. In c) some ants have found the nest by pure luck and have begun to form a trail. In d) the positive feedback process has resulted in the ants choosing the shortest route from the nest to the food source.

The pheromone found in nature is a chemical substance and this allows the pheromone to *evaporate*. This of course influences the behavior of the ants. A trail that is not traveled very often will eventually have no pheromone on it because of evaporation. This phenomenon reinforces the strength of the heavily traveled trails.

## 3.2 Ant colony systems for TSP

Probably inspired by the ants of nature Dorigo in 1991/1992 published his Ph.D. thesis [Dor92]. The thesis introduced the notion of ant systems as a way to solve optimization problems. To be more specific, the ant systems were used to find good solutions to the Traveling Salesman Problem (TSP).

The ant system will now be explained briefly. TSP is the problem of finding the shortest route between $n$ connected cities, where each city is only visited one time.

The distance between two cities $i$ and $j$ connected by a path is denoted $d_{ij}$. The path between cities $i$ and $j$ have an additional *trail value* denoted by $\tau_{ij}$. The trail value is the amount of pheromone on the path. All the ants are dispersed randomly among the $n$ cities.

Each ant selects a route based on the following algorithm. At the start city the ant chooses which city to move to based on a probability function. The probability function $p_{ij}$ weights the distance to the other cities and the amount of pheromone on the path to the city. If a city can not be reached in one step, the probability of choosing the path is 0.

$$p_{ij} = \frac{\tau_{ij} \cdot d_{ij}^{-\beta}}{\sum_{k \in V_i} \tau_{ik} \cdot d_{ik}^{-\beta}} \text{ where } V_i \text{ is the set of cities reachable from } i \text{ in one step.}$$

The ant moves to the chosen city and remembers the city that it just left by adding it to a *tabu list*.

When the ant has visited the $n$ cities, pheromone is added to the to tour just traveled. The amount added is constant and the longer the route the smaller the amount of pheromone added to each trail.

In order to prevent all the ants from choosing a particular strong trail, some of the pheromone on each trail is removed. The new trail value $\hat{\tau}_{ij}$ is $(1-\alpha) \cdot \tau_{ij} + \alpha \cdot \tau_0$. This corresponds to the evaporation of the pheromone spread by real ants.

Compared to the performance of other naturally inspired algorithms like genetic programming, neural networks, and simulated annealing, the ant colony system presented in [DG96] shows very good performance.

## 3.3 Ant colony systems for COPs

Probably inspired by the systems for TSP Bilchev and Parmee [BP95a, BP95b, BP95c, BP96] have introduced ant-inspired systems for optimization problems.

The TSP is a discrete problem in the sense that there are a finite number of possible solutions. The class of constrained optimization problems are continuous

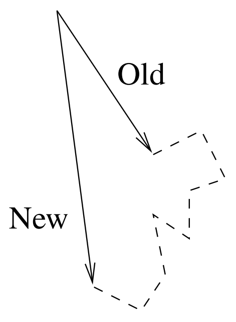

Figure 3.3: Example of an ant updating a path

problems with an infinite number of solutions. In the ant colony system for TSP, each ant chooses where to go using a probabilistic function and can only go to a finite set of destinations, i.e. the cities which are connected with the current city by a direct path. Bilchev and Parmee handles this in the ant colony system for constrained optimization problems by having a finite number of "destinations" called *paths*.

At the beginning these paths are distributed randomly in the search space. A path is just a point in the search space with an associated trail value and a function value.

As in the ant system for TSP, the ants choose a path probabilisticly based on the amount of pheromone associated with the paths. The amount of pheromone associated with each path is called the *trail value*. The ant performs local search with the end of the path as the starting point. The ant deposits an amount of pheromone proportional to the function value it has reached. If the ant discovers a new function value that is better than the function value of the end point of the path, then the end point of the path is moved to the position of the ant. This is illustrated in figure 3.3. The vector marked *Old* denotes the old path and the vector marked *New* denotes the path updated with the new position.

Furthermore, the paths can interact in what is called "random walk" and "trail diffusion" in the original papers. These processes will be explained in detail in the section about global search.

Last but not least, there is the process of evaporation. The pheromone evaporates during time and this means that the local search is guided away from paths which have not improved their fitness (function value).

At a more abstract level the ant system can be seen as a two-level system. There is the level of local search performed by the ants and there is the level of global search performed by the interacting paths. And perhaps most importantly, there is the communication between the two levels. The local search guides the global search when the paths are updated with the positions of the ants with improved function values. The other way around, the global search directs the local search with the amount of pheromone on each path. One possible advantage of this approach is, that it might have the right tradeoff between *exploration* and *exploitation*.

## 3.3.1 Global search

In the papers of Bilchev and Parmee [BP95a, BP95b, BP95c, BP96] the term *direction* is used to denote a possible starting point. The term *path* is used to denote a direction with a particular high pheromone value, ie a direction that attracts many ants. In this thesis no distinction is made between the two concepts — both are called paths regardless of pheromone value.

As explained in the next section about local search, the local search affects the paths. But the paths can also interact and create new paths. In the papers by Bilchev and Parmee two ways to do this are mentioned.

**Random walk**

The random walk is only mentioned briefly by Bilchev and Parmee. Wodrich has written a thesis about ant colony optimization and explains the random walk in detail based on his e-mail conversations with Bilchev.

*Random walk* is in fact just a specialized genetic algorithm. A new path is created by the following steps:

1. A random path is selected and the first entry in the new path vector is copied from the selected path. With a given probability, a new path is selected and the process is repeated until the new path vector has been created.

2. All entries in the new path vector are then mutated with a given probability by adding a small value to the entry in the vector.

The random walk differs from a ordinary genetic algorithm in a number of ways. Step one in the algorithm is very reminiscent of a normal crossover. A typical crossover uses two parents, whereas the “crossover” used by the random walk can have a variable number of parents ranging from one to the number of entries in the vector. Furthermore, the new selected paths are selected at random from all paths. The fitness of each path is not taken into account which is very unusual. Step two is just an ordinary mutation found in most genetic algorithms.

### Trail diffusion

*Trail diffusion* is very similar to the arithmetic crossover as used in many genetic algorithms. A new path is created using trail diffusion by the following steps:

1. Two paths are selected with equal probability amongst all paths.

2. The first entry in the new path vector is either:

   - The first entry in the vector of the first path.
   - The first entry in the vector of the second path.
   - The weighted average of the first entries of the vectors of both paths.

   The probability of the first two possibilities is 0.25 and the probability of choosing the weighted average is 0.50.

3. With a given probability, a new pair of paths are selected, otherwise the same pair of paths are used.

4. Steps 2 and 3 are repeated for subsequent entries in the vector of the new path.

A new path created by trail diffusion or random walk has not been traveled by ants and thus has a trail value of zero. In order to have a fair chance of attracting ants, the new path has to be assigned a trail value that is non-zero. This can be achieved by assigning a trail value that is the arithmetic average of the trail values of the paths which have contributed to the creation of the new path.

The number of paths are constant, so the newly created paths have to replace some of the existing paths. The *replacement* of new paths can be done in the same way that individuals are replaced when using steady-state genetic algorithms. Replacement can be done in the same way as the selection step in a genetic algorithm. One can for example use binary tournament replacement, where two individuals from the population are selected at random and the one with worst fitness is replaced [1]. Another even simpler possibility is to just replace the individual(s) with the worst fitness in the population.

Note that both trail diffusion and random walk select paths at random and thus there is no *selection pressure*. Instead there is what may be called *replacement pressure*. An ordinary steady-state genetic algorithms uses both selection and replacement and thus exhibits both kinds of pressure.

[1]Note that this implicitly results in an elite of size 1. The fittest individual will never loose a tournament and thus never get replaced.

#### Genetic algorithms

In this thesis *genetic algorithms* have been chosen for the creation of new paths. The concept of genetic algorithms is explained in the introduction. The details concerning the genetic algorithm are covered in section 3.3.4.

The most interesting thing concerning the choice of a genetic algorithm as the global search algorithm, is the choice of fitness function. A number of possibilities exist:

- The fitness is the function value at the end point of the path.
- The fitness is the trail value of the path.

The first possibility promotes paths with good function values. An ant starting its local search from a point in the search space with a good function value has a good chance of finding good function values in the neighborhood. Hence, this approach does not seem unreasonable.

The second possibility has some potential too. Paths with high trail values will be promoted and thus create new paths with high trail values. The high trail values characterize areas of the search space in which large improvements in the function values take place and thus makes this approach seem reasonable too.

Both fitness functions have been implemented and the results are presented in section 3.3.5.

### 3.3.2 Local search

The local search is implemented as a simple stochastic hill-climber. The stochastic hill-climbing process goes like this:

1. The ant selects a path with a probability proportional to the amount of pheromone on each path.
2. Using the end point of the path as a starting point for the ant, it chooses a direction to go from there.
3. If the previous ant improved the fitness of the path, the ant moves in the same direction. Otherwise it chooses a random direction.
4. The ant moves in the selected direction and the fitness in the new point is calculated.
5. If the function value of the new point is better than that of the path's old value, the path's end point is updated and an amount of pheromone proportional to the *improvement* of the fitness is added to the path.

It is important to realize that the choice of local search algorithm is not limited by the ant colony system. It is thus possible to choose a local search algorithm that is especially suitable for the specific problem ahead.

### 3.3.3 Introducing constraints

The ant colony system described so far has no mechanism to handle *constraints*. But with a few simple modifications, the ant colony system can handle constrained optimization problems as well.

On the level of global search, the constraints are handled by introducing selection pressure. When two individuals $i_1$ and $i_2$ are compared, it is done using the following rules:

1. If the constraint violation of $i_1$ is not equal to the constraint violation of $i_2$, the individual with the lowest constraint violation is the fittest.

2. If the two individuals have the same constraint violation, the one with the best function value is considered the fittest.

Note that according to the rules, a feasible individual is always considered fitter than an infeasible individual regardless of the function value. This generates a very high selection pressure towards feasible individuals.

On the level of local search the constraints are handled in an entirely different manner. When an ant discovers a point with a better function value the constraint violation must be below a certain *threshold.* The threshold has a high value at the beginning of the run and is lowered linearly to end at a value of zero in the last generation. This mechanism allow ants to sidetrack into the infeasible areas of the search space but as the threshold is lowered, the ants are forced into the feasible part of the search space.

Figure 3.4 shows an example of an ant sidetracking into the infeasible area of the search space. If the ant was restricted to the feasible area of the search space it would be much harder to get from A to B.

Furthermore, it has to be mentioned that the constraint-handling mechanisms does not *guarantee* a feasible solution.

### 3.3.4 The system

An ant colony system has been implemented for use in the experiments. The various parts of the ant colony system are described in the following sections.

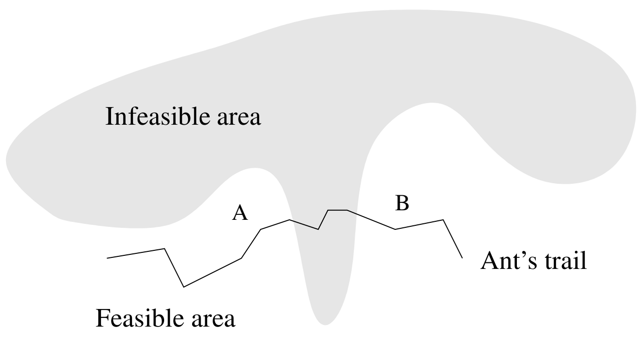

Figure 3.4: Example of an ant sidetracking into the infeasible area of the search space

## Global search

The global search is implemented using a steady-state genetic algorithm. Each path is represented as an individual by a real-valued vector.

The individuals used for recombination are selected by binary tournament selection and are recombined using uniform crossover. Each gene in the newly created individual is mutated with a given probability. The mutation adds a random number $r$ to the gene. The random number $r$ has a value between -10% and 10% of the domain for that gene. For the benchmark problem G6, the first variable $x_1$ has to satisfy $13 \leq x_1 \leq 100$ and this means that $r$ can have values in the interval $[-8.7, 8.7]$. As mentioned before, some kind of heuristics has to be used for assigning trail values to new individuals and in this case, this is done by averaging the trail values of the two parents.

By this procedure $n$ new individuals are created and these new individuals replace the $n$ individuals with the lowest fitness values.

## Local search

The local search algorithm used is a stochastic hill-climber. A stochastic hill-climber was chosen mainly because of its simplicity. It is very easy to implement and is computationally fast. Furthermore it is probably a good idea to investigate the performance of the ant colony system with a simple local search algorithm before going on to more sophisticated local search algorithms.

As described before, the ant chooses a path with a probability proportional to the trail value of the path. If the last ant improved the fitness of the path, the ant travels a random distance in the same direction as the previous ant. Otherwise, it chooses a random direction and travels a random distance. If the fitness of the new point is better than that of the path, the path is moved to the new point and the trail value is increased.

If the new point does not have a constraint violation lower than the current threshold, the point is rejected and the ant tries again. This is repeated a finite number of times (50) and if no point satisfying the threshold criteria is found, the ant gives up.

### 3.3.5 Experiments and results

Unless something else is mentioned, the following parameters have the same fixed values in all experiments reported.

- The number of paths is 200.
- The number of new paths for each generation is 80.
- The number of exploring ants is 20.
- The number of generations is 1000.

Even after 1000 generations some of the runs had not come up with a feasible solution, but this is just a small fraction of all the runs. To give the right picture only runs which came up with a feasible solution are used in the following.

#### Tuning the ant colony system

The first series of experiments was conducted to tune parameters like the crossover rate, the mutation rate and the evaporation rate. The mutation and crossover rates could have six different values: 0.0, 0.01, 0.05, 0.1, 0.5, or 1.0. The evaporation rate could have the following nine values: 0.1, 0.3, 0.6, 0.8, 0.85, 0.9, 0.95, 0.99, or 1.0. All combination of the values were tried, which sums up to 324 experiments. Each experiment was repeated ten times and the average values are used. The parameter settings providing the best results are reported in table 3.2.

Ten new runs were conducted with these parameters and the results are reported in table 3.3.

By looking at the experiment data a few conclusions can be drawn. For G8 the optimum value of 0.095825 was reached in all 10 runs in 55 of the 324 experiments and this is why no best parameters are reported. First of all, the best values for the mutation rates seem to be 0.1 and 0.05. The worst value seems to be 0.0. These values are not surprising. A value of 0.0 means no mutation at all and the best values of 0.05 and 0.1 indicate that the disruptive effect of the higher mutation rates is too large to provide good performance.

| Function | Mutation rate | Crossover rate | Evaporation rate |
|---|---|---|---|
| G1 | 0.10 | 0.50 | 0.95 |
| G2 | 0.10 | 0.50 | 0.85 |
| G4 | 0.05 | 0.50 | 0.99 |
| G6 | 0.00 | 0.50 | 0.60 |
| G7 | 0.10 | 0.50 | 0.99 |
| G8 | - | - | - |
| G9 | 0.10 | 0.50 | 0.99 |
| G10 | 0.50 | 1.00 | 0.99 |

Table 3.2: The best settings for the different functions.

| | | Optimum | Achieved |
|---|---|---|---|
| G1 | Best | -15 | -14.9971 |
| | Average | | -14.7885 |
| | Worst | | -12.9939 |
| G2 | Best | 0.803553 | 0.793070 |
| | Average | | 0.740943 |
| | Worst | | 0.643616 |
| G4 | Best | -30665.5 | -30665.4 |
| | Average | | -30664.9 |
| | Worst | | -30663.7 |
| G6 | Best | -6961.8 | -6928.3 |
| | Average | | -6898.3 |
| | Worst | | -6845.1 |
| G7 | Best | 24.306 | 24.666 |
| | Average | | 25.490 |
| | Worst | | 26.674 |
| G8 | Best | 0.095825 | 0.095825 |
| | Average | | 0.095825 |
| | Worst | | 0.095825 |
| G9 | Best | 680.63 | 680.88 |
| | Average | | 681.40 |
| | Worst | | 682.61 |
| G10 | Best | 7049.33 | 7317.41 |
| | Average | | 7432.74 |
| | Worst | | 7590.40 |

Table 3.3: The results of the experiments with the ant system

The best value for the crossover rate is 0.5 and the worst 0.0. The value of 0.0 corresponds to no crossover at all and this means that the crossover operator has a good influence on the performance.

Concerning the evaporation rate, no specific value gives superior performance. The general picture is that the higher values give the better performance. The value of 0.9 provides reasonable performance in most cases.

Furthermore, the experiments show that the mutation and crossover rates have a large impact on the performance, whereas the experiments are not very sensitive when it comes to the evaporation rate.

### The number of ants

To determine the impact of the use of ants, some experiments with the number of ants have been conducted. The functions G2, G4 and G9 have been chosen for experiments. All settings were the best ones from the previous experiments and the number of ants varied from 0 to 50. Figure 3.5 show the fitness of the fittest individual after 1000 generations versus the number of ants for the functions G2, G4, and G9.

The number of ants *does* have influence on the performance of the ant colony system for the function G2. The performance seem to get better and better with an increasing number of ants. On the other hand the performance of the ant colony system on the function G4 and function G9 shows no improvement when using more ants.

### Replacement strategies

In all the previous experiments the *delete worst* strategy has been used to find the individuals to replace. A number of different replacement strategies have been tried in a number of experiments.

- *Delete oldest* was introduced by De Jong and Sarma [DJS93]. As the name suggests, the oldest members of the population are replaced. Delete oldest is also called FIFO deletion.
- *Delete random* just deletes a random individual in the population.
- *Binary tournament replacement* replaces the less fit of two random individuals from the population.

For each replacement strategy ten runs were conducted with the parameters from table 3.2. Figure 3.6 shows the average fitness of the fittest individual for the different replacement strategies for the test functions G2, G4 and G9 with the number of generations along the $x$-axis.

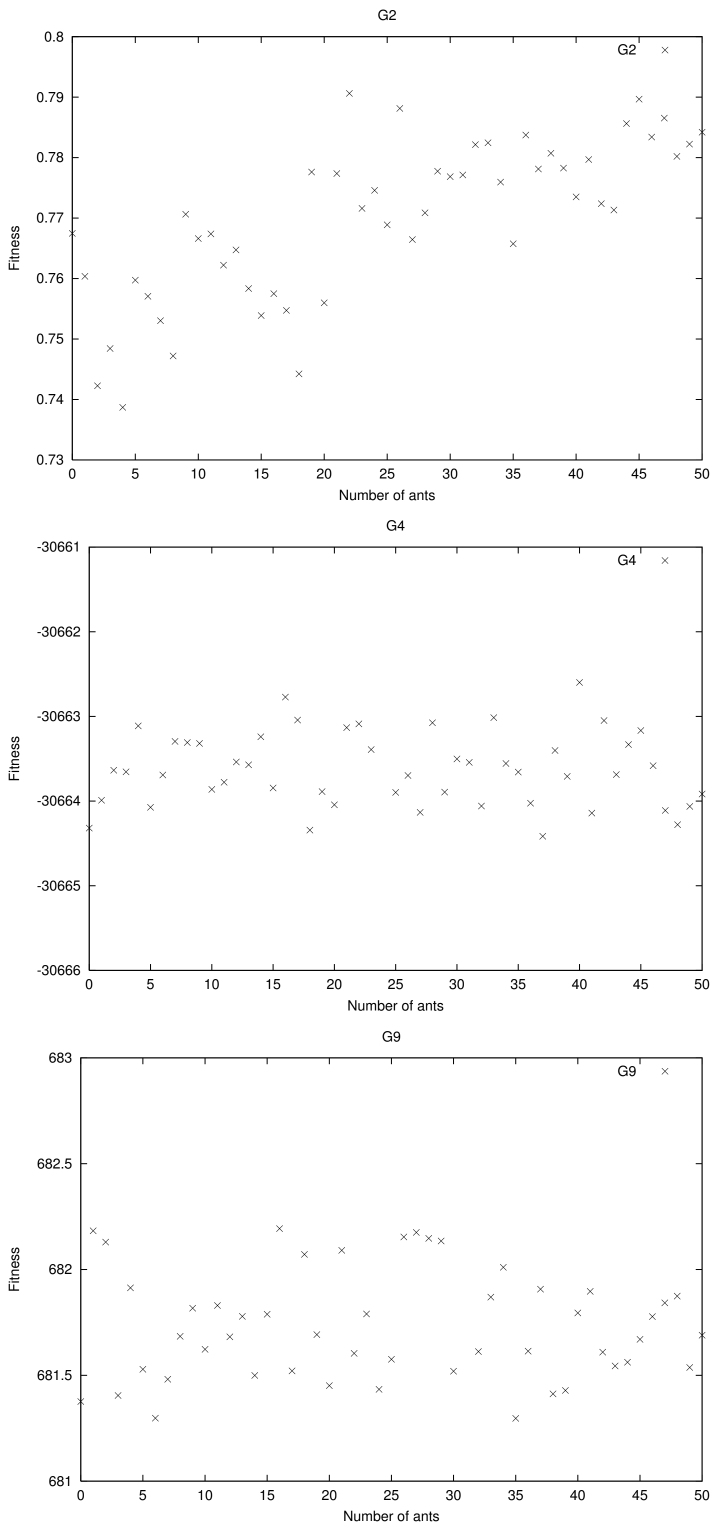

Figure 3.5: Fitness of fittest individual versus the number of ants - G2, G4, and G9.

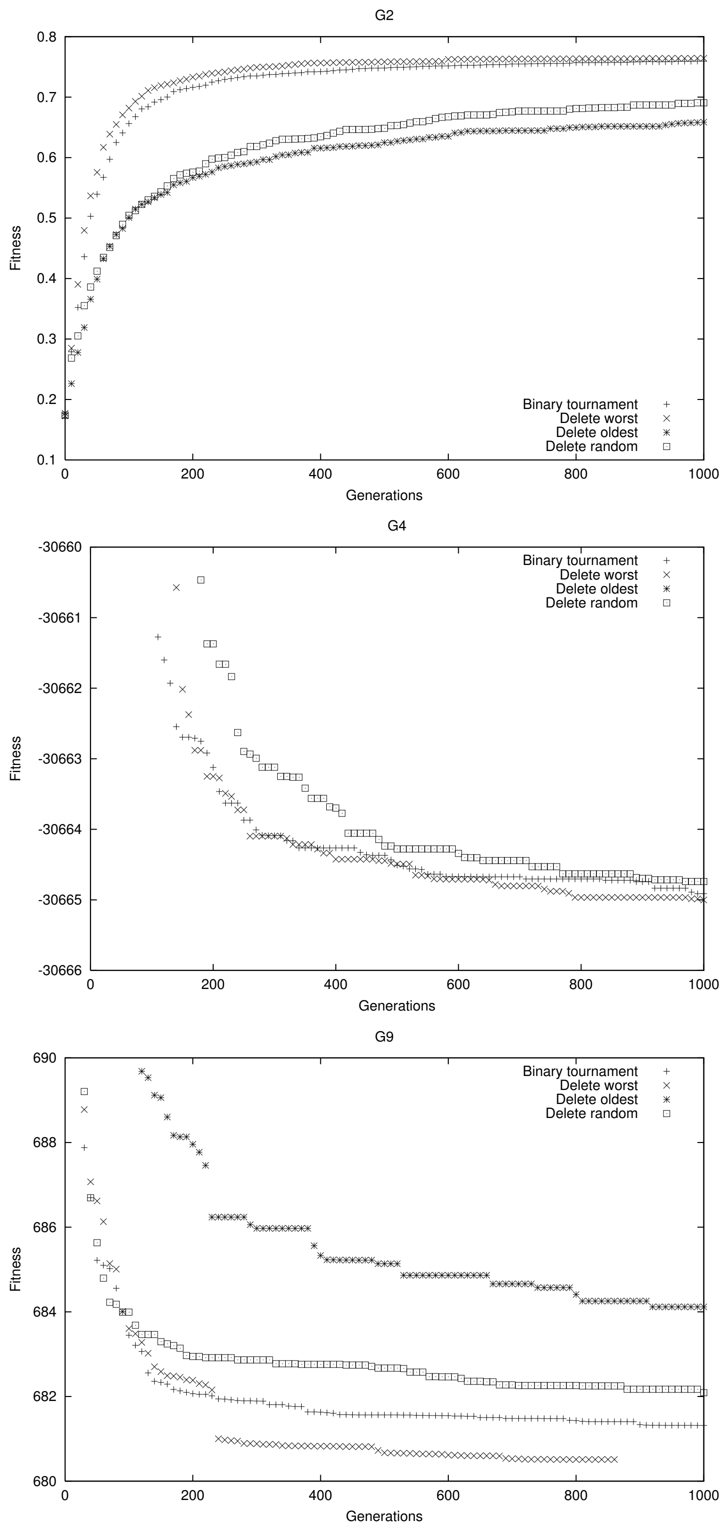

Figure 3.6: Fitness of fittest individual during time for the different replacement strategies - G2, G4, and G9.

On test function G2 the binary tournament and delete-worst strategies clearly outperforms the two other strategies. On test function G4 the three strategies of binary tournament, delete-worst, and delete-oldest have reached the same fitness value after 1000 generations. The performance of the delete-oldest strategy is so bad that it is not in the range of the plot. The plot of the performance of the replacement strategies on test function G9 shows the same picture. The delete-worst strategy is the best, closely followed by the binary tournament replacement strategy. The delete-random strategy has a better performance than the delete-oldest strategy. But then again, the delete-oldest strategy has the worst performance by far again.

All in all, the best replacement strategy is the delete-worst strategy and the worst strategy is the delete-oldest. Between the two other strategies, the binary tournament replacement strategy is the better.

The delete-oldest strategy deletes individuals without considering the fitness of the individuals. All individuals survive only for a fixed number of generations. This means that the fit individuals *always* will get replaced and this might explain the bad performance of the delete-oldest strategy. In the delete-random strategy a good individual at least has *some* chance of not getting deleted. The two other approaches favor fit individuals. The delete-worst strategy does this by not deleting good individuals. The binary tournament replacement strategy also does this by not deleting the best of two individuals. In fact, the fittest individual will never get deleted as it can not loose a binary tournament (being fitter than the rest of the individuals).

### Fitness function

The selection in the global search favors individuals with high fitness. The fitness function is thus of utmost importance — it is sometimes said, that finding the right fitness function is half the solution to a problem. In all previous experiments a fitness function based on the constraint violation and the function value was used.

Another possibility is to somehow incorporate the trail value of the path into the fitness function. The trail value of a path is a measure of how good the path is as a start point for the exploring ants. Hence, it does not seem unreasonable to use this knowledge in the fitness function. There are a number of possibilities:

- Fitness is the function value: $f(\vec{x}) = f_{obj}(\vec{x})$.
- Fitness is the trail value: $f(\vec{x}) = \tau$.
- Fitness is a linear combination of the function value and the trail value: $f(\vec{x}) = a \cdot f_{obj}(\vec{x}) + b \cdot \tau$.

| a | b | G1 | G2 | G4 |
|---|---|---|---|---|
| 0.0 | 1.0 | -9.8461 | 0.170717 | -30407.9 |
| 1.0 | 0.0 | -14.9871 | 0.751441 | -30664.8 |
| 1.0 | 0.01 | -14.6623 | 0.786504 | -30664.9 |
| 1.0 | 0.05 | -14.9797 | 0.766640 | -30665.1 |
| 1.0 | 0.1 | -14.9856 | 0.772002 | -30665.1 |
| 1.0 | 0.5 | -14.8602 | 0.761876 | -30665.1 |
| 1.0 | 1.0 | -14.5104 | 0.762429 | -30664.9 |
| 1.0 | 5.0 | -14.8694 | 0.778417 | -30665.1 |
| 1.0 | 10 | -14.6527 | 0.766896 | -30664.7 |
| 1.0 | 50 | -14.8712 | 0.758819 | -30665.1 |
| 1.0 | 100 | -14.6532 | 0.763607 | -30664.8 |
| 1.0 | 500 | -14.6644 | 0.756049 | -30665.1 |
| 1.0 | 1000 | -14.8642 | 0.783252 | -30665.0 |
| 1.0 | 5000 | -14.7114 | 0.777750 | -30665.0 |
| 1.0 | 10000 | -14.8430 | 0.751031 | -30664.9 |

Table 3.4: Results of runs with different linear combinations.

Actually, the two first possibilities are just special cases of the last possibility with $a = 1.0, b = 0.0$ and $a = 0.0, b = 1.0$ respectively.

A number of experiments have been run to investigate the influence of the variables $a$ and $b$. As in the previous experiments, the parameters from table 3.2 were each used in ten runs and the average fitness of the fittest feasible individual is reported. Table 3.4 shows the performance for the different settings of $a$ and $b$ and test functions G1, G2, and G4.

Depending solely on the trail value in the fitness function does not seem to be a good idea, as can be seen from the results of $a = 0.0$, $b = 1.0$. The remaining results do not give a clear picture of whether it is a good idea to incorporate trail value in the fitness function or not. Further research in this area is needed.

## 3.4 Real-life problems

Real-life problems with more than one constraint are characterized by the fact that different constraints have different priorities. In optimization problems this is reflected by hard and soft constraints.

The ant system handle constraints on the global search level by selecting individuals according to their constraint violation. If two individuals have equal constraint violation, the function value is taken into consideration. On the level of local search, the ants are allowed to have a constraint violation below a threshold. This threshold is lowered during the run.

One possible way to deal with the problem of hard and soft constraints is to make a hybrid of the ant system and the penalty weight approach. The hard constraints are handled in the same way as the constraints were handled in the ordinary ant system. The soft constraints are handled by adding a penalty to the fitness just like it is done with the decoder approach.

This is just one of the ways to make the ant system handle hard and soft constraints. Future research of this and other methods seems to be a quite promising and could be very interesting.

# Chapter 4

# Using a Decoder for COPs

*Caterpillar to Butterfly: "How do you become a butterfly?"*
*Butterfly: "You have to be willing to die."*
*Caterpillar: "Die?"*
*Butterfly: "Well, it feels like you're dying. But it really turns out to be a transformation to something better."*
Unknown

## 4.1 The decoder approach

In 1998 Kozieł and Michalewicz [MK98a, MK98b] introduced a new way of handling constraints. The idea is to use a decoder to map the feasible region of the search space to an $n$-dimensional cube and then use an ordinary genetic algorithm to search for the optimum.

### 4.1.1 The decoder and its properties

When a genetic algorithm is normally used, there is a direct correspondence between the genotype and the phenotype. Suppose we want to optimize the function $f(\vec{x}) = x_1 + x_2$. The normal way to do this is to let each individual in the genetic algorithm have two genes corresponding to $x_1$ and $x_2$ respectively, and use the genotype as the phenotype when evaluating the individual.

Instead of having this direct mapping from genotype to phenotype, one could imagine a more complex mapping. This mapping is called a *decoder*. The phenotype is created by decoding the genotype. In the above example, the genes $g_1$ and $g_2$ are mapped to $x_1$ and $x_2$. An example of a decoder could be to have the following mapping $x_1 = g_1/g_2$ and $x_2 = g_1 + g_2$. Another more complex example is the use of decoder in the solution of TSP problems. The genotype contains directions

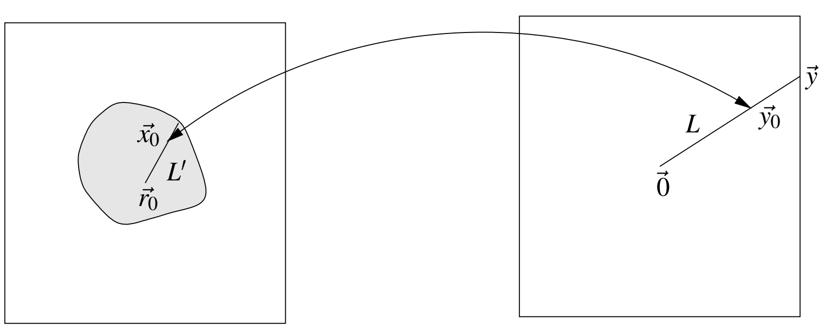

Figure 4.1: Mapping between the feasible region and the hypercube

on how to build the phenotype (ie a route) using the decoder. The solution is built in a way that always yields a feasible solution.

The idea introduced by Kozieł and Michalewicz was to make a mapping between the feasible region of the search space and an $n$-dimensional hypercube. The decoder uses this mapping to map individuals with genotypes belonging to the hypercube to points in the feasible regions of the search space. The decoder must satisfy several conditions [1]

- For each feasible solution $f \in \mathcal{F}$ there exists an encoded solution $e$.
- For each encoded solution $e$ there exists a feasible solution $f$.
- All solutions in $\mathcal{F}$ should be represented by the same number of encodings.

In other words, there should be a one-to-one correspondence between the encoded solutions and points in the feasible region, and all points in the feasible region should have a corresponding encoded solution.

In order to be useful, the mapping must be computationally fast. Furthermore, the mapping must have a locality feature such that small changes in the encoded solution corresponds to small changes in the feasible solution.

### 4.1.2 A simple decoder

Let us start by looking at a search space with one convex feasible region. This is the situation in figure 4.1 with a 2-dimensional search space and a 2-dimensional hypercube (a square).

We would like to find a mapping from the hypercube to the feasible region of the search space that satisfies the requirement from section 4.1. Given a point $\vec{y}_0$

[1]See [MK98b], p. 5.

in the hypercube, we would like to find the corresponding point $\vec{x}_0$ in the feasible region of the search space.

An arbitrary point $\vec{y}_0$ ($\vec{y}_0 \neq \vec{0}$) in the hypercube $[-1,1]^n$ defines a line segment $L$ from $\vec{0}$ to the boundary of the hypercube. The point on the boundary is denoted $\vec{y}$. A single entry $y_i$ in the vector $\vec{y}$ is:

$$y_i = y_{0,i} \cdot t, \text{ for } i = 1,\dots,n \text{ where } t \in [0, t_{\max}]$$

and $t_{\max} = 1/\max\{|y_{0,1}|,\dots,|y_{0,n}|\}$. Clearly, if $t = 0$ then $\vec{y} = \vec{0}$, and if $t = t_{\max}$ then $\vec{y} = (y_{0,1} \cdot t_{\max},\dots,y_{0,n})$ which is a boundary point of the hypercube.

The next step is to transform the line segment $L$ in the hypercube to the corresponding line segment $L'$ in feasible region of the search space. For this a *reference point* $\vec{r}_0$ is needed. The reference point is just an arbitrary point in the feasible region.

For an arbitrary point $\vec{y'}$ on $L$, the corresponding point $\vec{x'}$ on $L'$ is given by:

$$\vec{x'} = \vec{r}_0 + \vec{y}_0 \cdot \tau \text{ where } \tau = \tau_{\max}/t_{\max}$$

and $\tau_{\max}$ has a value such that $\vec{r}_0 + \vec{y}_0 \cdot \tau_{\max}$ is a boundary point of the feasible search space. And indeed, for $\vec{0}$ the corresponding point is $\vec{r}_0$ and for $\vec{y}$ the corresponding point is on the boundary of the feasible region. And of course:

$$\vec{x}_0 = \vec{r}_0 + \vec{y}_0 \cdot \tau$$

The mapping described has all the properties that we wanted from the decoder. There is one-to-one correspondence between the encoded solutions and points in the feasible region. And all points in the feasible region have a corresponding encoded solution. Furthermore, being linear, the mapping has the locality feature and is computationally fast.

### 4.1.3 A not so simple decoder

The simple mapping defined in the previous section is only valid if the line segment starting in $\vec{r}_0$ intersects the boundary of the feasible region in one place. This means that it is a requirement that the feasible region is convex, and that there exists only one feasible region. This is seldom the case for any realistic constrained optimization problem.

This is why a new mapping now will be defined. This mapping handles nonconvex feasible regions as well as the case of more than one feasible region. The mapping is depicted in figure 4.2

Let us start by defining of mapping $g$ between the $n$-dimensional hypercube and the search space $\mathcal{S}$. The mapping $g : [-1,1]^n \mapsto \mathcal{S}$

$$g(\vec{y}) = \vec{x} \text{ where } x_i = y_i \frac{u(i) - l(i)}{2} + \frac{u(i) + l(i)}{2} \text{ for } i = 1,\dots,n$$

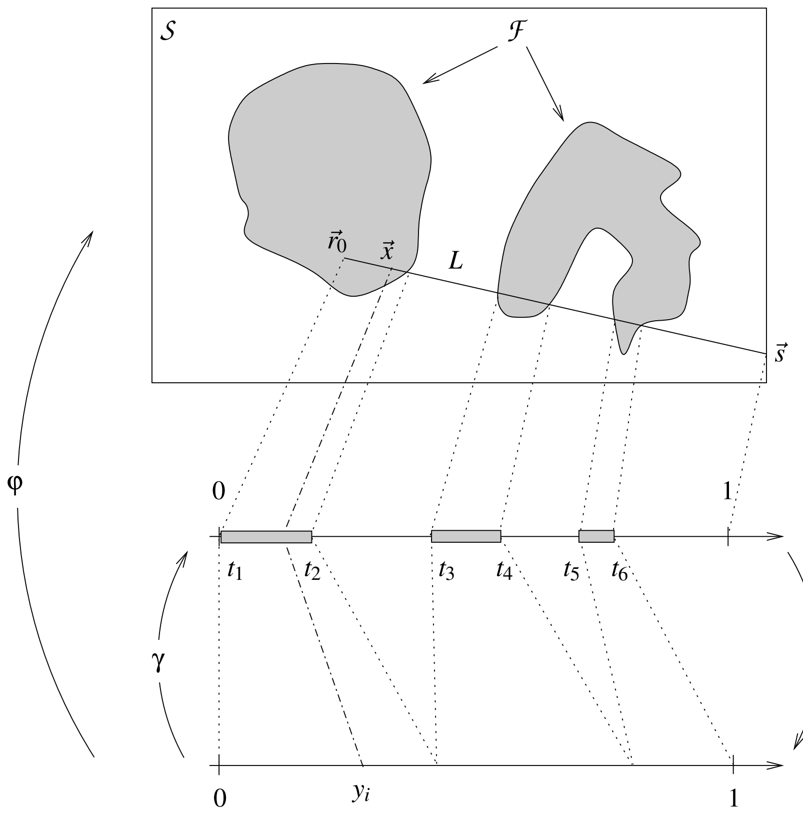

Figure 4.2: Example of mapping between the interval $[0,1]$ and the feasible regions of the search space

One can think of the function $g$ as a mapping that stretches the hypercube in $n$ dimensions to fit the $n$-dimensional rectangle defined by the domains of the variables.

As in the case of the simple decoder, we start by looking at a point $\vec{y} \in [-1,1]^n$ which we would like to decode to the corresponding point $x_0 \in \mathcal{F}$. Together with a reference point $\vec{r}_0$ which again just is a arbitrary point in the feasible region $\mathcal{F}$, $\vec{y}$ defines a line segment $L$ from $\vec{r}_0$ to a point $\vec{s} = g(\frac{\vec{y}}{y_{\max}})$ on the boundary of the search space $\mathcal{S}$.

$$L(\vec{r}_0, \vec{s}) = \vec{r}_0 + t \cdot (\vec{s} - \vec{r}_0) \text{ for } 0 \le t \le 1$$

It is clear that in the case of a search space with one convex feasible region there exists $t_0$ such that for $t \in [0, t_0]$ the corresponding points on the line segment $L$ are in the feasible region and for $t = t_0$, the point is on the boundary of the feasible region.

If there is more than one feasible region or the feasible region is non-convex, the line segment $L$ may intersect the boundary of $\mathcal{F}$ in more than one point. This can be explained by looking at figure 4.2. Imagine starting at $\vec{r}_0$ and traveling along $L$ towards $\vec{s}$. On the way the boundaries of the feasible regions will be crossed a number of times and the corresponding values of $t$ define the feasible intervals.

In more formal terms there may be several intervals of feasibility

$$[t_1, t_2], \ldots, [t_{2k-1}, t_{2k}]$$

There are in other words $k$ intervals where $t_i$ together with $L$ define points on the boundary of $\mathcal{F}$.

In order to have a mapping from the hypercube to the feasible regions of the search space, we have to define an additional mapping from $[0,1]$ to $\bigcup_{i=1}^{k}[t_{i-1}, t_i]$. Instead of having a mapping from $[0,1]$ to $\bigcup_{i=1}^{k}[t_{i-1}, t_i]$, a mapping $\gamma$ from $]0,1]$ to $\bigcup_{i=1}^{k}]t_{i-1}, t_i]$ is defined. This change makes it possible to concatenate the intervals making the mapping one-to-one and the left boundary point of each interval can still be calculated with arbitrary precision.

The following is depicted in the lower part of figure 4.2. The mapping $\gamma$ is defined as follows:

$$\gamma(a) = t_{2j-1} + d_j \frac{a - \delta(t_{2j-1})}{\delta(t_{2j}) - \delta(t_{2j-1})}$$

where

- $\delta(t) = (t - t_{2i-1} + \sum_{j=1}^{i-1})/d$ where $d_j = t_{2j} - t_{2j-1}, d = \sum_{j=1}^{k}$ and $t_{2i-1} \leq t \leq t_{2i}$
- $\delta(t_{2k}) < a$ if $k \leq j$

The intuition behind the mapping is to concatenate the intervals so that the mapped intervals have a length proportional to their length divided by the sum of the lengths of all intervals. It can be easily verified that the mapping $\delta$ is the reverse of the mapping $\gamma$.

Now the mapping $\varphi$ can be defined. $\varphi$ maps an arbitrary point $\vec{y}$ in the hypercube $[-1,1]^n$ to the corresponding point $\vec{x}$ in the feasible region.

$$\varphi(\vec{y}) = \begin{cases} \vec{r}_0 + t_0 \cdot (g(\vec{y}/y_{\max}) - \vec{r}_0) & \text{if } \vec{y} \neq \vec{0} \\ \vec{r}_0 & \text{otherwise} \end{cases}$$

where $r_0$ is the reference point, ie a point in the feasible region $\mathcal{F}$, $t_0 = \gamma(|y_{\max}|)$ and $y_{\max} = \max |y_i|$ for $1 \leq i \leq n$.

To sum it up:

- A mapping is established between the line segment $L$ and $[0,1]$.
- Another mapping is established between the subintervals of $[0,1]$ corresponding to the feasible parts of $L$ and $[0,1]$. The subintervals are in a way enlarged and glued together to cover all of $[0,1]$.
- Together the two mappings form $\varphi$ which is a one-to-one mapping between $[-1,1]^n$ and $\mathcal{F}$.

Being one-to-one, the mapping satisfies *some* of the requirements of a good decoder. Compared to the simple decoder in the previous section, the new decoder lacks the locality feature in points close to the boundary of two feasible regions. This may be a problem as the optimum solution often are located on the boundary of the feasible region. When certain requirements are satisfied, the mapping is computationally fast with the calculation of the feasible intervals $[t_1,t_2],\ldots,[t_{2k-1},t_{2k}]$ being the hardest part.

### 4.1.4 Finding the feasible intervals

We want to find the feasible intervals

$$[t_1,t_2],\ldots,[t_{2k-1},t_{2k}]$$

One way to do this, is to search the interval $[0,1]$ linearly but this is not exactly a fast method. Another way is to divide the interval $[0,1]$ into a number (e.g. 200) of subintervals and then use binary search to find the $t_i$'s. The pseudo code for the binary search algorithm is shown in figure 4.3. This method is a lot faster than doing linear search, but has the drawback that it assumes that there is only one value of $t_i$ in each subinterval. If this assumption does not hold, the two situations depicted in figure 4.4 could occur. In a) an infeasible region is marked as feasible and in b) no values are mapped to one of the feasible regions.

In order to minimize this problem, one can choose a larger number of intervals and furthermore look at the constraints one at a time. If we look at one constraint at a time we may have a situation like in figure 4.5. Looking at one constraint at a time and using binary search, all intervals for $c_1$ and $c_2$ are found and they can then be used to find the intervals that satisfy both constraints. If we were considering all constraints at once, the intervals $[t_3,t_4]$ and $[t_5,t_6]$ would not have been found.

```
function BinarySearch(Vector Start, Vector Stop): Vector
begin
    Mid := (Start + Stop)/2
    if (|Mid - Start| > Threshold) then
    begin
        if (Feasible(Mid) $\neq$ Feasible(Stop)) then
        begin
            Mid := BinarySearch(Mid, Stop)
        end
        else
        begin
            Mid := BinarySearch(Start, Mid)
        end
    end
    return Mid
end
```

Figure 4.3: Binary search pseudo code

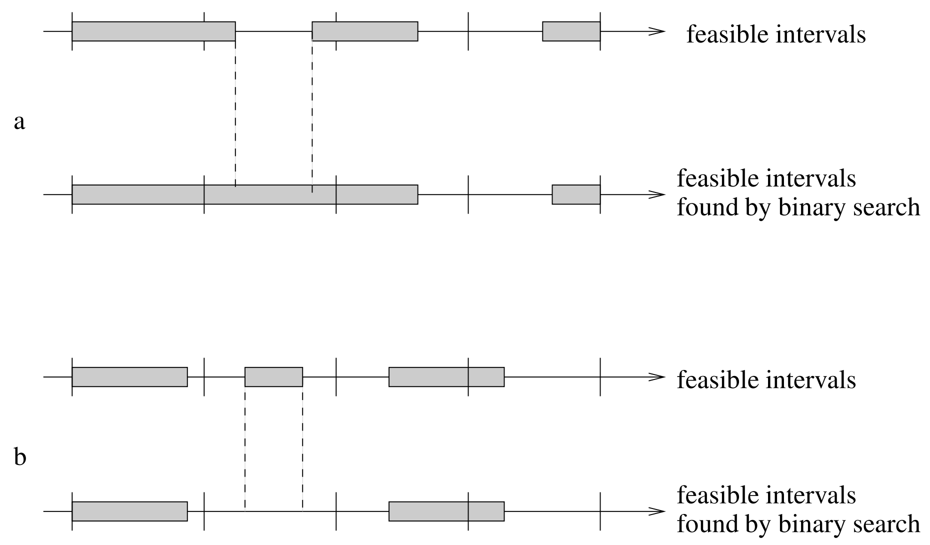

Figure 4.4: Possible errors when using division into intervals

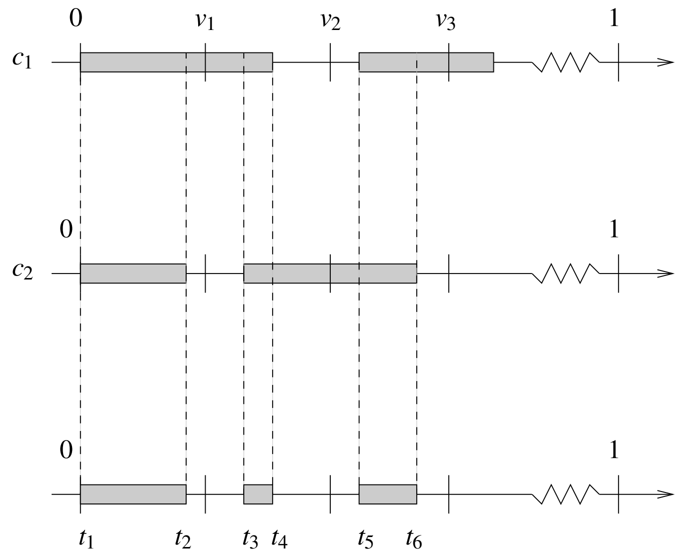

Figure 4.5: The union of the intervals of constraints $c_1$ and $c_2$

### 4.1.5 Properties of the decoder approach

#### Advantages

The decoder approach has several advantages. First of all, no extra parameters are needed. In GA a lot of time is used to find the right settings for the parameters for the problem ahead. In the penalty weight approach there are parameters like the penalty weight (constant weight), the start and stop values of the penalty weight (deterministic changing weights) and so on. The ant system has even more parameters.

Another advantage of the decoder approach is that it always yields a feasible solution and this is guaranteed. There is no such guarantee for the penalty weight approach and for the ant system.

Another thing that may be in favor of the decoder approach, is that it spends all the time searching just the feasible region. It does not waste any time evaluating infeasible solutions and this may give a speedup in performance.

#### Disadvantages

The decoder approach requires a feasible solution for $r_0$ and this may be a problem, as for some problems it is very hard to find a feasible solution. In fact the whole class of problems called *constrained satisfaction problems* is about finding feasible solutions. The way it is handled in the system is to do random search for a feasible point in the search space and this may not be the most effective approach. Furthermore, the performance of the system depends on $r_0$ as will be seen in section 4.4.1, so a random point might even in some cases decrease the performance.

Another potential problem is that for problems with convex feasible regions and/or more than one feasible region. the locality feature may not be present. In the case of convex and/or more that one feasible regions, the hypercube consists of several regions that are “glued together”. This means that two points that are close in the hypercube may be very far away from each other in the search space. The missing locality feature makes the problem harder to solve for the genetic algorithm.

Finding the boundaries of the feasible regions is sometimes computationally hard. For simple problems with few feasible regions this is not noticeable but for more complex problems the computation times grow.

The last problem is that there is no obvious way to handle hard and soft constraints. For the benchmark problems all the constraints are hard in the sense that they must be satisfied for a solution in order for the solution to be feasible. For a problem with hard and soft constraints it is of course possible to construct the hypercube with the encoded solutions using the hard constraints. The soft constraints make some of the regions of the hypercube more attractive than others due to a smaller violation of the soft constraints. An extension for handling hard and soft constraints is proposed in the section 4.5.

## 4.2 The system

To test the decoder approach, a system has been built. This system makes it possible to test the performance of the decoder approach on the eight benchmark problems. With the exception of the decoder part, the system is just an ordinary genetic algorithm. The features of the genetic algorithm will be described in the following part.

The genetic algorithm uses real-valued representation. The genetic algorithm in the two papers by Michalewicz and Kozieł uses binary representation and according to Michalewicz [2] the decoder approach has not been implemented with floating point representation before.

The selection mechanism is binary tournament and an elite size of one is used. Recombination of the selected parents is done using distributed crossover, where each gene comes from one particular parent with a probability of 0.5. The offspring created using the uniform crossover is mutated by means of biased muta-

[2]Private communication

| Function | Mutation rate | Crossover rate | Elite size |
|---|---|---|---|
| G1 | 0.50 | 1.00 | 1 |
| G2 | 0.10 | 1.00 | 1 |
| G4 | 0.10 | 0.50 | 0 |
| G6 | 0.50 | 1.00 | 1 |
| G7 | 0.10 | 1.00 | 1 |
| G8 | 0.50 | 0.50 | 1 |
| G9 | 0.10 | 1.00 | 1 |
| G10 | 0.10 | 0.50 | 1 |

Table 4.1: The best parameters for each function using the decoder approach

tion. Using biased mutation, a random number is added to each gene selected for mutation. The random number is generated using a normal distribution with mean 0 and standard deviation 0.1.

As in all the other genetic algorithms in this thesis, the number of individuals in the population is set to 30 for no particular reason.

All genes have values between -1 and 1. When the fitness of an individual is calculated, the chromosome is decoded yielding a point $\vec{x}$ in the search space and the fitness corresponding to $\vec{x}$ is assigned to the individual.

## 4.3 Experiments and results

### 4.3.1 Fine tuning

First, a number of experiments were run to determine the best settings of the mutation rate, crossover rate and the elite size. The following values were used for the mutation rate and the crossover rate: 0.0, 0.01, 0.10, 0.50, and 1.00. The elite size had values of 0 and 1. This amounts to 50 experiments. Each experiment was run ten times to minimize uncertainty related to the stochastic nature of the algorithm. Furthermore, the experiments were performed using the binary search method described in section 4.1.4.

For each test function the best settings are displayed in table 4.1.

No particular set of parameters is the best on all the benchmark problems, but there are some patterns. First of all, the mutation rate should not be too small neither too large. A mutation rate of 0.10 seems to be a good compromise. The crossover rate should have a value close to the maximum of 1.0. If the choice is between 0.0, 0.01, 0.10, 0.50, and 1.00, the values 0.50 and 1.0 should be preferred. In almost all cases it seems to be a good idea to have an elite size of 1.

The best general settings for the benchmark problems are a mutation rate of 0.10, a crossover rate of 1.0 and an elite size of 1.

The performance of the algorithm is heavily affected by the parameter values and if possible, one should always try to fine tune the parameters to a particular problem. The settings provided in the previous paragraph may be a good starting point in a fine tuning effort.

The next series of experiments were done using the parameters in table 4.1. Due to limitations of the binary search algorithm mentioned in section 4.1.5 the decoder sometimes returned an infeasible point in the search space. This was discovered as the system sometimes produced results that were better than the optimum! Each time an infeasible solution was returned, the point was recalculated using linear search to fix this problem. This method was used in producing the results displayed in table 4.2

The decoder approach has quite good performance for all the benchmark problems. The results were especially good for G6 but not so good for G1 compared to the other approaches. Better results for G1 are reported in [MK98b] so systems based on the decoder approach *are* able to achieve good (but not outstanding) results for G1. The feasible region(s) of G1 could have very complex shape(s) and mapping them into the hypercube might make the problem harder.

## 4.4 Other issues

In the papers by Michalewicz and Kozieł [MK98a, MK98b] some further investigations have been made. These topics are discussed in this section.

### 4.4.1 Changing $r_0$ during the run

In all experiments so far, the reference point $r_0$ has remained the same during the run. But maybe it could improve performance if the reference point was changed dynamically during the run. In [MK98a] some experiments were conducted where the reference point was changed a fixed number of times during the run. The reference point was moved to the location of the best solution found so far. Moving the reference point changes the phenotypes of all genotypes and to have the same phenotypes in the population the genotypes have to be recalculated. Therefore one may or may not reevaluate all individuals when changing $r_0$.

The experiments conducted by Michalewicz and Kozieł show that for some problems it is beneficial to change the reference point and for some it is not. When it comes to the reevaluation of the genotypes this feature does not seem to be important in terms of performance. So in order to have the best possible

| | | Optimum | Achieved |
|---|---|---|---|
| G1 | Best | -15 | -14.8274 |
| | Average | | -12.9097 |
| | Worst | | -6.3201 |
| G2 | Best | 0.803553 | 0.803166 |
| | Average | | 0.787440 |
| | Worst | | 0.759830 |
| G4 | Best | -30665.5 | -30665.2 |
| | Average | | -30663.5 |
| | Worst | | -30659.3 |
| G6 | Best | -6961.8 | -6961.5 |
| | Average | | -6961.5 |
| | Worst | | -6961.4 |
| G7 | Best | 24.306 | 24.531 |
| | Average | | 26.205 |
| | Worst | | 29.331 |
| G8 | Best | 0.095825 | 0.095825 |
| | Average | | 0.095825 |
| | Worst | | 0.095825 |
| G9 | Best | 680.63 | 680.63 |
| | Average | | 680.66 |
| | Worst | | 680.74 |
| G10 | Best | 7049.33 | 7200.37 |
| | Average | | 7849.97 |
| | Worst | | 9499.22 |

Table 4.2: Results for the decoder approach

performance one should try to tune the number of changes in the reference point and not bother too much with the reevaluation of genotypes.

### 4.4.2 Other mappings

The decoding of the individuals in the hypercube is very simple. The mapping used in the decoding process does not have to be this simple and may be arbitrarily complex. One of the ways to change the mapping is to introduce an additional mapping

$$\omega(\vec{y}) = \vec{y'} \text{ where } y'_i = a \cdot y_i$$

The mapping $\omega$ maps vectors from the hypercube $[-1,1]^n$ to the hypercube

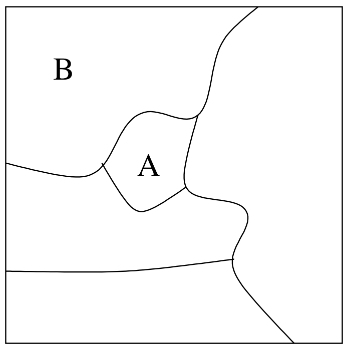

Figure 4.6: The hypercube with different feasible regions

with the same dimensions in a non-uniform way. The value of $a$ has the following meaning

- For a vector $\vec{y}$ large values of $a$ will result in $\vec{y'}$, which is farther away from the reference point and closer to the boundary of the feasible region than $\vec{y}$.
- Correspondingly, small values of $a$ will result in $\vec{y'}$ being closer to the reference point and farther away from the boundary of the feasible region than $\vec{y}$.

The optimum solution of a constrained optimization problem is often located on the boundary of one of the feasible regions. Large values for $a$ tend to concentrate the search in the regions around the edges of the hypercube corresponding to a some of the boundaries of the feasible regions. Figure 4.6 shows an example of a 2-dimensional hypercube with several different feasible regions mapped into it. If the optimum solution is located in region B near the edge of the hypercube, large values of $a$ might turn out to be a good idea. But on the other hand, if the optimum solution is located on the boundary of region A, large values of $a$ may not be a good idea.

Small values of $a$ tend to concentrate the search around the reference point which may or may not have a positive influence on the performance.

In the article [MK98a] some reports of experiments with the following mapping is presented:

$$\omega(\vec{y}) = \vec{y'} \text{ where } y' = y_i \cdot y_{\max}^{k-1}$$

The experiments have been done with the values 0.0, 1.0 and 3.0 for $k$. For most experiments the value 3.0 turned out to be the best. This high value of $k$ corresponds to a small value of $a$ and thus the search effort is concentrated in the region around the reference point.

## 4.5 Future research

One of the problems addressed in this chapter, is that the decoder approach has no way of handling hard and soft constraints.

A possible way to handle the problem of hard and especially soft constraints is to make a *hybrid* of the decoder approach and the penalty weight approach. The hybrid would work in the following way:

- Use the hard constraints to construct the encoding to the hypercube.
- Penalize the solutions proportionally to their violation of the soft constraints. This makes it possible to give different penalty weights to different soft constraints.

The decoder part of the hybrid guarantees that a solution that satisfies the hard constraints is found. The penalty weight part will contribute in finding solutions with minimal violation of the soft constraints.

# Chapter 5

# Conclusion and Comparison of the Three EA Techniques

In this final chapter a summary of the three techniques will be presented, followed by a comparison of the performance with a detailed description for each test case.

## 5.1 Summary of the techniques

### 5.1.1 The penalty weight approach

The penalty weight approach works by adding a penalty to the fitness function of infeasible individuals according to the following rule:

$$f_{pen}(\vec{x}) = f_{obj}(\vec{x}) + W \cdot P'(\vec{x})$$

The work of Eigen, Hinterding, and Michalewicz [EHM98] with respect to adaptation of parameters in evolutionary algorithms suggests that the parameters can be grouped in four categories:

- Constant parameters
- Dynamic parameters
  - Deterministic changing parameters
  - Adaptive parameters
  - Self-adaptive parameters

With this as inspiration, a system with the four ways of handling the penalty weight was built. In the past, systems with constant penalty weights, with deterministic changing penalty weights, and adaptive penalty weights have been built and experiments have been performed. Experiments with the *new* approach of self-adapting the penalty weights were performed and compared to similar experiments with the three other approaches.

The experiments showed that when carefully fine tuned, the deterministic changing penalty weight scheme had the best performance. The second best method was in fact the new approach of self-adapting the penalty weights. This method had better performance than the methods of constant penalty weights and the adaptive penalty weights which is quite encouraging. The results of the experiments with the deterministic changing weights are reported in table 5.1.

Furthermore, a number of experiments were conducted to determine the role of the mutation rate and the size of the initial penalty weights with respect to the performance of the self-adapting penalty weight scheme. For most test cases, the size of the mutation rate and the initial weight did not affect the performance. The performance on the remaining test cases was affected by the parameters and thus the method of self-adaptation does not remove the need of adjusting the parameters of the algorithm to the problem.

### 5.1.2 The ant system

Dorigo [DG96] introduced a system for the Traveling Salesman Problem inspired by ants foraging for food. A system for optimization problems also inspired by ants was introduced by Bilchev and Parmee [BP95a, BP95b, BP95c, BP96].

There are three levels in the ant system:

**The local search level** is the level where the single ants do local search with the end of a path as a starting point. The ants are able to deposit and detect pheromone and use this as a means to communicate with the other ants.

**The global search level** is the level of the paths. A path is just a point in the search space with an associated trail value. New paths may be created from old paths using trail diffusion, random walk, genetic algorithms or a completely different global search strategy.

**The intermediate level** is the level of communication between the two other levels. When the ants find a point that is better than the path it started from, the path is updated to the new point.

The system used for the experiments utilized a stochastic hill-climber for the local search and a steady-state genetic algorithm for the global search.

A number of experiments were performed to tune the value of the mutation rate, the crossover rate and the evaporation rate. With the best settings found ,the experiments were repeated and the results are reported in table 5.1.

Furthermore, a series of experiments with the number of ants have been performed. The experiments showed that the performance of the system is sensitive to the number of ants for some of the test problems. The system uses a steady-state genetic algorithm and thus a number of experiments with different replacement strategies were performed. The results of the experiments made it evident that the delete-worst strategy was the most successful strategy.

The trail value of a path may give a good indication of how good the path is as a starting point for exploration performed by the ants. With this in mind, one can think of several ways of using this information in the fitness function of the genetic algorithm. Experiments were performed with several linear combinations of the objective function and the trail value as the fitness function. These experiments showed that a fitness function relying solely on the trail value had bad performance. In the remaining cases there is no clear picture and this is an interesting subject for future research.

### 5.1.3 The decoder approach

In most genetic algorithms, there is a direct correspondence between the genotype and the phenotype. The mapping between the genotype and the phenotype can nonetheless be of arbitrary complexity.

In [MK98a, MK98b] Michalewicz, Kozieł introduced the intriguingly simple idea of using a decoder to map genotypes in an $n$-dimensional hypercube to phenotypes in the feasible region(s) of the search space. The following mapping is used to map $\vec{y} \in [-1,1]^n$ to $\vec{x} \in \mathcal{F}$:

$$\varphi(\vec{y}) = \begin{cases} \vec{r}_0 + t_0 \cdot (g(\vec{y}/y_{\max}) - \vec{r}_0) & \text{if } \vec{y} \neq \vec{0} \\ \vec{r}_0 & \text{otherwise} \end{cases}$$

The decoder has been implemented in a system which uses a genetic algorithm with real-valued representation. In [MK98b] a system using Gray coding was used for experiments with good results. According to the article similar or better results are expected for a system using real-valued representation and the results in this thesis confirm that this claim is indeed true.

Some fine tuning experiments were performed and the results are reported in table 5.1. Furthermore, the role of the reference point $r_0$ and other mappings was discussed.

## 5.2 Comparison of the three techniques

Often, when a new technique is proposed in an article, the seemingly superior performance is demonstrated by looking at the performance on a few test cases. The results are then compared to that of some traditional techniques on the same test cases. This may not be completely fair, as the test cases may be very well suited to this particular new technique. Furthermore, the author quite understandably has often spent a considerable amount of time on his/her new technique and know good settings for the test cases. Often the same amount of time has not been used to carefully tune the traditional techniques. This may of course affect the result, making the comparison not completely fair.

With this in mind, some good advice is:

- Use several test cases with different properties when it comes to the dimensionality, the complexity of the function and the constraints, the topology of the search space, and the size of the feasible regions compared to the size of the search space.
- Spend some time tuning the techniques to the test cases.

The test cases in this thesis are of the form mentioned in the first advice. Furthermore, considerable effort went into tuning the parameters of the different techniques. Hopefully this has resulted in a fair comparison.

### 5.2.1 The test cases

Different properties of the test cases are displayed in table 5.2. LI is the number of Linear Inequalities, NI is the number of Non-linear Inequalities, AC is the number of Active Constraints and the column Ratio is the ratio of feasible individuals in the population.

A program has been created to give an idea of the complexity of the feasible region with respect to the number of feasible regions and the non-convexity of the feasible regions.

The program works by creating random points in the search space until two feasible points have been found. The next step is to investigate the feasibility of the points on the line between the two feasible points. This investigation can have two results:

- All points are feasible.
- Some points are infeasible.

This procedure was repeated 1000 times. The number of times that the investigation yielded infeasible points on the line between the two feasible points are also reported in table 5.2 (the column denoted 'Complexity').

Figure 5.1 is an constructed example of the purpose of the program. The figure shows the feasible regions for a problem with several non-convex feasible

| | | Penalty | Ants | Decoder | Optimum |
|---|---|---|---|---|---|
| G1 | Best | **-14.9997** | -14.9971 | *-14.8274* | 15.0000 |
| | Average | **-14.9990** | -14.7885 | *-12.9097* | |
| | Worst | **-14.9982** | -12.9939 | *-6.3201* | |
| G2 | Best | *0.784245* | 0.793070 | **0.803166** | 0.8036 |
| | Average | *0.732685* | 0.740943 | **0.787440** | |
| | Worst | 0.670220 | *0.643616* | **0.759830** | |
| G4 | Best | -30665.3 | **-30665.4** | *-30665.2* | -30665.5 |
| | Average | -30664.5 | **-30664.9** | *-30663.5* | |
| | Worst | -30661.1 | **-30663.7** | *-30659.3* | |
| G6 | Best | *-6922.8* | -6928.3 | **-6961.5** | -6961.8 |
| | Average | *-6893.4* | -6898.3 | **-6961.5** | |
| | Worst | -6865.0 | *-6845.1* | **-6961.4** | |
| G7 | Best | *24.880* | 24.666 | **24.531** | -24.306 |
| | Average | 25.755 | **25.490** | *26.205* | |
| | Worst | 27.213 | **26.674** | *29.331* | |
| G8 | Best | 0.095825 | 0.095825 | 0.095825 | 0.095825 |
| | Average | 0.095825 | 0.095825 | 0.095825 | |
| | Worst | 0.095825 | 0.095825 | 0.095825 | |
| G9 | Best | 680.72 | *680.88* | **680.63** | 680.63 |
| | Average | 680.90 | *681.40* | **680.66** | |
| | Worst | 681.45 | *682.61* | **680.74** | |
| G10 | Best | - | 7317.41 | **7200.37** | 7049.33 |
| | Average | - | **7432.74** | 7849.97 | |
| | Worst | - | **7590.40** | 9499.22 | |
| Best | | 3 | 7 | 11 | |
| Worst | | 8 | 5 | 8 | |

Table 5.1: Results for the three approaches.

regions. Lines have been drawn between pairs of random points in the feasible regions. Line *A* starts in one feasible region and ends in another and thus there are infeasible points on the line. Line *B* also contains infeasible points but this is due to the non-convexity of the feasible region. The lines *C* and *D* do not cross the boundaries of the feasible regions. The program would report that 2 out of 4 lines contain infeasible points.

| Function | Dimensions | LI | NI | AC | Ratio | Complexity |
|---|---|---|---|---|---|---|
| G1 | 13 | 9 | 0 | 6 | 0.000180 | 0 |
| G2 | 20 | 0 | 2 | 1 | 99.996964 | 152 |
| G4 | 5 | 0 | 6 | 1 | 26.950348 | 41 |
| G6 | 2 | 0 | 2 | 2 | 0.006760 | 0 |
| G7 | 10 | 3 | 5 | 6 | 0.000110 | 0 |
| G8 | 2 | 0 | 2 | 0 | 0.864150 | 23 |
| G9 | 7 | 0 | 4 | 2 | 0.523310 | 7 |
| G10 | 8 | 3 | 3 | 6 | 0.000560 | 0 |

Table 5.2: Properties of the test functions.

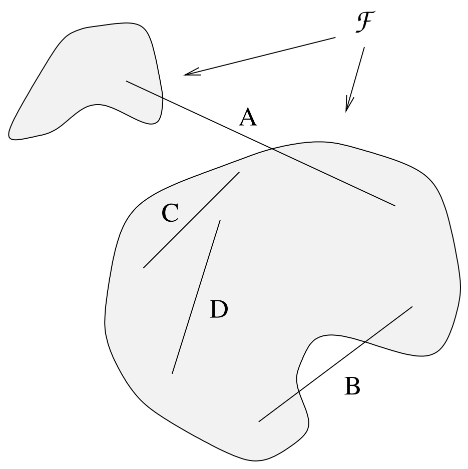

Figure 5.1: Example

### 5.2.2 The performance on the test cases

**G1**

The G1 function is quadratic and has nine linear constraints. None of the lines between the 1000 random pairs of feasible points contained infeasible points and this suggests that there is just one non-convex feasible region in the search space. The penalty approach and the ant system achieve results very close to the optimum with the penalty approach being the best. The decoder approach is not very good on G1 for some reason. Maybe the mapping between the hypercube and the feasible region actually makes the problem harder to solve and this might explain the bad performance of the decoder approach.

**G2**

The G2 function is nonlinear and the search space has 20 dimensions. There a two constraints - one linear and the other polynomial. The complexity number is 152 and this means that the feasible regions are either non-convex and/or the number of feasible regions is high. This is one of the harder test problems and the decoder approach clearly outperforms the two other approaches. The worst result in the ten runs of the decoder approach is in fact better than the average results of the two other techniques.

**G4**

The G4 function is quadratic and the six constraints are quadratic. All three techniques have good performance on this function with the ant system having the best performance by a small margin.

**G6**

The G6 function is cubic and there are two quadratic constraints which both are active at the optimum. The complexity number is 0 which suggests that there is one non-convex feasible region in the search space. The performance of the penalty approach and the ant system was good. Again the decoder approach outperformed the other approaches with a clear margin. The decoder approach consistently found values very close to the optimum.

**G7**

The G7 function is quadratic and the search space has 10 dimensions. There are three linear constraints and five quadratic constraints. The feasible part of the search space is only a small fraction of the search space, and in fact it has the smallest ratio of all test cases. Combined with the fact that six constraints are active at the optimum this problem seems to be very hard. The performance of the three techniques is approximately the same with the ant system having a slightly better performance.

**G8**

The G8 function is nonlinear and has two quadratic constraints of which none are active at the optimum. This problem turned out to be very easy for all techniques and the optimum was reached in all 10 runs with all three techniques.

**G9**

The G9 function is polynomial and has four quadratic constraints with two of them being active at the optimum. All three techniques had good performance on the problem with the decoder approach being slightly better than the other two.

**G10**
The G10 function is linear and has three linear constraints and three quadratic constraints and *all* are active at the optimum. The complexity number of 0 suggests that there is only one convex feasible region. This problem turned out to be very hard. The penalty approach was not even able to find a feasible solution. The decoder approach gave the best result but the ant system provided the best results on average.

The test functions have very different properties and it is hard, based on the test cases, to predict what technique will have the best performance on a given problem. It seems though, that the decoder approach has good performance on some problems which are tough for the penalty approach and vice-versa. Take a look at for example G1, G2, and G6.

### 5.2.3 The final verdict

All three techniques have been fine tuned and tried on a number of test cases. The results show that *no* single technique is superior on *all* test cases. Furthermore, all techniques had acceptable performance on all problems which just emphasizes how important it is to tune the parameters of the technique to the problem.

The decoder approach *did* have the best results in 11 out of 21 possible categories and for some test cases the results were much better than the results of the other techniques. If one is to pick a winner amongst the three techniques, the decoder approach would take the price.

The ant system had good and stable performance on all test cases and therefore the second place goes to the ant system. The penalty weight approach had acceptable performance for most test problems. There were problems with G2 and the approach failed completely on the G10 test function.

On the other hand, the penalty weight approach is very easy to implement, is computationally fast and also handles hard and soft constraints easily. These are also factors one should consider when choosing a technique for a constrained optimization problem and each technique has its advantages and disadvantages.

# Bibliography

[Bäc92] Thomas Bäck, *The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm*, Parallel Problem Solving from Nature (Amsterdam) (Männer and B. Manderick, eds.), vol. 2, Elsevier, 1992, pp. 85–94.

[Bäc97a] Thomas Bäck, *Mutation parameters*, p. E1.2.1:7, In Bäck et al. [BFM97], 1997.

[Bäc97b] Thomas Bäck, *Self-adaptation*, p. C7.1.1:15, In Bäck et al. [BFM97], 1997.

[BESS98] Thomas Bäck, Agoston E. Eigen, Marc Schoenauer, and Hans-Paul Schwefel (eds.), *Proceedings of the 5th conference on parallel problems solving from nature*, Lecture Notes in Computer Science 1498, New York, NY, Springer Verlag, 1998.

[BFM97] Thomas Bäck, David Fogel, and Zbigniew Michalewicz (eds.), *Handbook of evolutionary computation*, Institute of Physics Publishing Ltd, Bristol and Oxford University Press, New York, 1997.

[BP95a] George Bilchev and Ian C. Parmee, *Adaptive search strategies for heavily constrained design spaces*, Proceedings of the 22nd International Conference CAD'95, Ukraine, Yalta, 8-13 May 1995, 1995.

[BP95b] George Bilchev and Ian C. Parmee, *The ant colony metaphor for searching continuous design spaces*, Proceedings of the AISB Workshop on Evolutionary Computation, University of Sheffield, UK, 3-4 April 1995, 1995.

[BP95c] George Bilchev and Ian C. Parmee, *Natural self-organizing systems*, Tech. Report Internal Report No PEDC-01-95, Plymouth Engineering Design Centre, 1995.

[BP96] George Bilchev and Ian C. Parmee, *Constrained optimization with an ant colony search model*, Proceedings of ACEDC'96, 1996.

[DG96] Marco Dorigo and Luca Maria Gambardella, *Ant colonies for the traveling salesman problem*, Tech. Report TR/IRIDIA/1996-3, Univerité Libre de Bruxelles, 1996.

[DJS93] Kenneth A. De Jong and Jayshree Sarma, *Generation gaps revisited*, Foundations Of Genetic Algorithms 2 (L. Darrel Whitley, ed.), Morgan Kaufmann Publishers, 1993, pp. 19–28.

[Dor92] Marco Dorigo, *Optimization, learning and natural algorithms*, Ph.D. thesis, Politecnico di Milano, Italy, 1992.

[EHM97] Agoston E. Eigen, Robert Hinterding, and Zbigniew Michalewicz, *Adaptation in evolutionary computation: A survey*, In Proceedings of the 4th IEEE Conference on Evolutionary Computation [Pro97], pp. 65–69.

[EHM98] Agoston E. Eigen, Robert Hinterding, and Zbigniew Michalewicz, *Parameter control in evolutionary algorithms*, Tech. Report 98-07, Leiden University, 1998.

[Esh97] Larry J. Eshelman, *Genetic algorithms*, p. B1.2.1:11, In Bäck et al. [BFM97], 1997.

[EvdH97] A. E. Eiben and J. K. van der Hauw, *Solving 3-sat with adaptive genetic algorithms*, In Proceedings of the 4th IEEE Conference on Evolutionary Computation [Pro97], pp. 81–86.

[FP90] C.A. Floudas and P.M. Pardalos, *A collection of test problems for constrained global optimization problems*, Lecture Notes in Computer Science 455, Springer-Verlag, 1990.

[Him72] David M. Himmelblau, *Applied nonlinear programming*, McGraw-Hill, 1972.

[Hin97] Robert Hinterding, *Self-adaptation using multi-chromosomes*, In Proceedings of the 4th IEEE Conference on Evolutionary Computation [Pro97], pp. 87–91.

[JH94] Jeffrey A. Joines and Christopher R. Houck, *On the use of non-stationary penalty functions to solve non-linear constrained optimization problems with ga's*, Proceedings of the First IEEE Conference on Evolutionary Computation, vol. 2, 1994.

[KP98] Spyros Kazarlis and Vassilios Petridis, *Varying fitness functions in genetic algorithms: Studying the rate of increase of the dynamic penalty terms*, In Bäck et al. [BESS98].

[MA94] Zbigniew Michalewicz and N. Attia, *Evolutionary optimization of constrained problems*, Proceedings of the 3rd Annual Conference on Evolutionary Programming (River Edge, NJ) (A. V. Sebald and L. J. Fogel, eds.), World Scientific Publishing, 1994, pp. 98–108.

[Mic95] Zbigniew Michalewicz, *Genetic algorithms + data structures = evolution programs*, third ed., Springer-Verlag, 1995.

[Mit96] Melanie Mitchell, *An introduction to genetic algorithms*, MIT Press, 1996.

[MK98a] Zbigniew Michalewicz and Sławomir Kozieł, *A decoder-based evolutionary algorithm for constrained parameter optimization problems*, In Bäck et al. [BESS98].

[MK98b] Zbigniew Michalewicz and Sławomir Kozieł, *Evolutionary algorithms, homomorphous mappings, and constrained parameter optimization*, To appear in Evolutionary Computation (1998).

[MS96] Zbigniew Michalewicz and Marc Schoenauer, *Evolutionary algorithms for constrained parameter optimization problems*, Evolutionary Computation **4** (1996), no. 1, 1–32.

[Pro97] *Proceedings of the 4th ieee conference on evolutionary computation*, IEEE Press, 1997.

[SC97] Alice E. Smith and David W. Coit, *Penalty functions*, p. C5.2.1:6, In Bäck et al. [BFM97], 1997.

[SS97] Martin Schmidt and Thomas Stidsen, *Hybrid systems: Genetic algorithms, neural networks, and fuzzy logic*, Trykkeriet, Matematisk Institut, Aarhus Universitet, 1997.

[TR98] Andrew Lawrence Tuson and Peter Ross, *Adapting operator settings in genetic algorithms*, Accepted for publication in Evolutionary Computation, 1998.

[Tus95] Andrew Lawrence Tuson, *Adapting operator probabilities in genetic algorithms*, Master's thesis, Department of Artificial Intelligence, Edinburgh University, U.K, 1995.

[Whi93] Darrel Whitley, *A genetic algorithm tutorial*, Tech. Report CS-93-103, Colorado State University, 1993.

[WM95] David H Wolpert and William G. Macready, *No free lunch theorems for search*, Tech. Report SFI-TR-95-02-010, The Santa Fe Institute, 1995.

[Wod96] Marc Wodrich, *Ant colony optimisation - an empirical investigation into an ant colony metaphor for continuous function optimisation*, University of Cape Town, 1996.

# Appendix A

# Installing and Running the Programs

## A.1 Programs for the penalty approach

Download the package from
`http://www.daimi.au.dk/~banger/thesis/penalty.tar.gz`

Uncompress it to a directory using
`tar penalty.tar.gz xvf - | gunzip`

Compile the program using the provided makefile:
`make sga`

### A.1.1 Program options

**sga** [ *options* ]

*Options*

| | |
|---|---|
| **-cr** *crossover rate* | Use *crossover rate* as the crossover rate for the genetic algorithm. E.g. `-cr 0.80`. The default value is 1.0 |
| **-mu** *mutation rate* | Use *mutation rate* as the mutation rate for the genetic algorithm. E.g. `-mu 0.01`. The default value is 0.1 |
| **-mn** *mutation mean* | Use *mutation mean* as the mean value in the normal distribution used in mutation. Example `-mn 0.0`. The default value is 0.0 |

| | |
|---|---|
| **-sd** *mutation standard deviation* | Use *mutation standard deviation* as the standard deviation of the normal distribution used in mutation. Example `-sd 0.1`. The default value is 0.5 |
| **-po** *population size* | Use *population size* to set the number of individuals in the population. Example `-po 25`. The default population size is 30 |
| **-el** *elite size* | Use *elite size* to set the size of the elite used in the genetic algorithm. For example `-el 2` specifies an elite of size 2. The default elite size is 0. |
| **-ge** *number of generations* | Use *number of generations* to set the number of generations. Example `-ge 1000`. The default number of generations is 100. |
| **-G** *function number* | Specify *function number* to decide which function the algorithm should optimize. For example `-G 2`. The default function is G1. |
| **-con** *weight* | When this parameter is specified the genetic algorithm uses a constant penalty weight with the value given by *weight*. For example `-con 1000`. |
| **-det lin** *start-weight stop-weight* | Use this parameter to specify that the genetic algorithm should use a deterministic changing penalty weight. The weight is changed linearly with start value *start-weight* and stop value *stop-weight*. An example of this is `-det lin 100 1000`. |
| **-det exp** *stop-weight* | Use the parameter to specify that the genetic algorithm should use a deterministic changing penalty weight. The weight is changed in an exponential fashion with 0 as the start weight and *stop-weight* as the value of the weight at the end of the run. For example `-det exp 1000`. |

| | |
|---|---|
| **-ada abs** *delta threshold* | Use this option to make the penalty weight adaptive. If the ratio of feasible individuals is bigger than *threshold*, the value *delta* will be added to the global weight and otherwise subtracted. For example `-ada abs 1 0.10` will add 1 to the global weight when the number of feasible individuals is larger than 10% of the whole population. |
| **-ada rel** *delta ratio* | Use this option to make the penalty weight adaptive. If the number of feasible individuals is bigger than *ratio*, the global weight will be increased by *delta* i.e. $W' = W \cdot (1 + \delta)$ and otherwise decreased. For example `-ada rel 0.1 0.10` will increase the value of the global weight by 1% when the number of feasible individuals is larger than 10% of the whole population. |
| **-sad uni** *initial weight* | Use this option to make the penalty weight self-adaptive. The penalty weights of the individuals are set using a uniform distribution with mean *initial weight*. |
| **-sad nor** *initial weight* | Use this option to make the penalty weight self-adaptive. The penalty weights of the individuals are set using a normal distribution with mean *initial weight*. |
| **–max** | When this option is specified, the largest of the two penalty weights are used when two individuals are compared in connection with the self-adaptive penalty weight approach. |
| **–min** | When this option is specified, the smallest of the two penalty weights are used when two individuals are compared in connection with the self-adaptive penalty weight approach. |
| **–avg** | When this option is specified, the average value of the two penalty weights are used when two individuals are compared in connection with the self-adaptive penalty weight approach. |

| | |
|---|---|
| **-wmu** *weight-mutation rate* | Use this option when using self-adaptive penalty weights to set the mutation rate of the penalty weight gene to the value *weight-mutation rate*. For example one could specify `-wmu 0.01`. The default value is 0.10. |
| **-wcr** *weight-crossover rate* | Use this option when using self-adaptive penalty weights to set the crossover rate of the penalty weight gene to the value *weight-crossover rate*. For example `-wcr 0.80`. The default value is 1.0. |
| **--silent** | Use this option to suppress some of the information printed by the program. All information is printed on lines starting with a hash mark (`#`) |
| **--plot** | Use `--plot` to plot the results of the run in columns separated by tabular spaces. This format is readable by the statistics program. |

## A.2 Programs for the ant system

Download the package from
`http://www.daimi.au.dk/~banger/thesis/ants.tar.gz`

Uncompress it to a directory using
`tar ants.tar.gz xvf - | gunzip`

Compile the program using the provided makefile:
`make aco`

### A.2.1 Program options

**aco** [ *options* ]

*Options*

| | |
|---|---|
| **-G** *function number* | Specify *function number* to decide which function the algorithm should optimize. For example `-G 1`. The default function is G2. |
| **-re** *number of paths* | Use the **-re** option to specify the number of paths (the number of individuals in the population). For example `-re 100`. The default number of paths is 200. |
| **-lo** *number of ants* | The number of ants can be specified using the **-lo** option. For example `-lo 40`. The default number of ants is 20. |
| **-gl** *number of new paths* | The genetic algorithm is a steady-state genetic algorithm and this option allows one to set the number of new individuals created in each generation. For example `-gl 40`. The default value is 80. |
| **-ev** *evaporation rate* | The default value of the evaporation rate is 0.90. The **-ev** option can be used to set this value, for example `-ev 0.95`. |

| | |
|---|---|
| **-ge** *number of generations* | The number of generations is specified using the **-ge** option. In each generation a number of individuals are created and they replace existing individuals. With the default values 80 new paths are created each generation. The system runs for 1000 generations with `-ge 1000`. |
| **-cr** *crossover rate* | The crossover rate for the genetic algorithm can be specified by using the **-cr** option. For example `-cr 0.80`. The default crossover rate is 1.0. |
| **-mu** *mutation rate* | The mutation rate can be set to a value different from the default value of 0.5 by using the **-mu** option. For example `-mu 0.10`. |
| **--binarytournament** | Use this option to use the binary tournament replacement strategy when replacing existing individuals by new ones. The default replacement strategy is the delete worst strategy. |
| **--deleterandom** | Use this option to use the delete random replacement strategy. |
| **--deleteoldest** | Use this option to use the delete oldest replacement strategy. |
| **--deleteworst** | Use this option to use the delete worst replacement strategy. This is the default replacement strategy and is used when no other replacement strategies are specified. |

## A.3 Programs for the decoder approach

Download the package from
`http://www.daimi.au.dk/~banger/thesis/decoder.tar.gz`

Uncompress it to a directory using
```
tar decoder.tar.gz xvf - | gunzip
```

Compile the program using the provided makefile:
```
make constraints
```

### A.3.1 Program parameters

**constraints** [ *options* ]

*Options*

| | |
|---|---|
| **-cr** *crossover rate* | Use *crossover rate* as the crossover rate for the genetic algorithm. E.g. `-cr 0.80`. The default value is 1.0 |
| **-mu** *mutation rate* | Use *mutation rate* as the mutation rate for the genetic algorithm. E.g. `-mu 0.01`. The default value is 0.1 |
| **-mn** *mutation mean* | Use *mutation mean* as the mean value in the normal distribution used in mutation. Example `-mn 0.0`. The default value is 0.0 |
| **-sd** *mutation standard deviation* | Use *mutation standard deviation* as the standard deviation of the normal distribution used in mutation. Example `-sd 0.1`. The default value is 0.5 |
| **-po** *population size* | Use *population size* to set the number of individuals in the population. Example `-po 25`. The default population size is 30 |
| **-el** *elite size* | Use *elite size* to set the size of the elite used in the genetic algorithm. For example `-el 2` specifies an elite of size 2. The default elite size is 0. |
| **-ge** *number of generations* | Use *number of generations* to set the number of generations. Example `-ge 1000`. The default number of generations is 100. |

| | |
|---|---|
| **-G** *function number* | Specify *function number* to decide which function the algorithm should optimize. For example `-G 2`. The default function is G1. |
| **-v** *number of intervals* | Use the **-v** option to set the number of intervals used in the binary search routine. The default value is 20 and another number can be specified by for example `-v 100`. |
| **-lin** | Use the **-lin** option to use linear search instead of binary search when searching for the boundaries of the feasible regions. This option makes the program accurate but also very slow. |
| **-binlin** | Use the **-binlin** option to perform search for the boundaries of the feasible regions using a hybrid of binary and linear search. Normally, binary search is used, but when an infeasible point is returned, the point is recalculated using linear search. |
| **--silent** | Use this option to suppress some of the information printed by the program. All information is printed on lines starting with a hash mark (`#`) |
| **--plot** | Use `--plot` to plot the results of the run in columns separated by tabular spaces. This format is readable by the statistics program. |

## A.4 Additional programs

Download the package from
`http://www.daimi.au.dk/~banger/thesis/misc.tar.gz`

Uncompress it to a directory using
`tar misc.tar.gz xvf - | unzip`

### A.4.1 Ratio of feasible vs. infeasible regions

Compile the program using the provided makefile:
`cd misc; make feasible`

The program **feasible** calculates the ratio of feasible points in the search space, by creating a number of random points in the search space and checking if they are feasible or not. Example of output from the program:

```
G1      Feasible 2      Infeasible 999998       Ratio 0.000200
G2      Feasible 999965 Infeasible 35           Ratio 99.996498
G4      Feasible 269699 Infeasible 730301       Ratio 26.969900
G6      Feasible 65     Infeasible 999935       Ratio 0.006500
G7      Feasible 1      Infeasible 999999       Ratio 0.000100
G8      Feasible 8621   Infeasible 991379       Ratio 0.862100
G9      Feasible 5137   Infeasible 994863       Ratio 0.513700
G10     Feasible 3      Infeasible 999997       Ratio 0.000300
```

Note that in this example only 1.000.000 points are used. 10.000.000 points were used to calculate the numbers used in table 1.1 in the thesis.

### A.4.2 The complexity of the feasible regions

Compile the program using the provided makefile:
`cd misc; make convex`

The program **convex** finds 1000 random pairs of points in the feasible regions, connects them with lines and reports the number of lines with infeasible points. This number gives an idea of how complex the feasible region is with respect to convexity and the number of feasible regions.

### A.4.3 Shell scripts and programs for statistics

The `misc` directory also contains a number of shell scripts used in the fine tuning experiments. Furthermore, there are a number of programs used to process the results of the experiments.

# Appendix B

# Information about this thesis

## B.1 Programs

This thesis was created using `Emacs 19.34.1` and `pslatex 3.14159`. All figures were created using `Xfig 3.2` and the graphs were created using `gnuplot 3.7` and `Matlab 5.3.0.10183`. The Postscript version of the thesis was created from the dvi-file using `dvips(k) 5.78`. The PDF-file was created from the Postscript file with the `ps2pdf` command.

## B.2 Downloading the thesis

The thesis can be downloaded in compressed Postscript format from
`http://www.daimi.au.dk/~banger/thesis/thesis.ps.gz`

and in Portable Document Format (PDF) from
`http://www.daimi.au.dk/~banger/thesis/thesis.pdf`

Both are in the A4 paper format. The PDF-file has the additional advantage that it can be searched.

## B.3 Contacting the author

The author of this thesis — Jørgen Bang Erichsen — can be reached at the following email addresses: `banger@daimi.au.dk` or `jorgen.bang@erichsen.net`. All questions are welcome.